Desarrollo de un motor de inferencia para RIF en Python

Adrián Tormos Llorente

Director: Javier Béjar Alonso
(Departamento de Ciencias de la Computación)

Grado de Ingeniería Informática
Especialidad de Computación
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) — BarcelonaTech

23 de enero de 2020

Abstract

Este trabajo de fin de grado consiste en el desarrollo de un sistema en el lenguaje Python que realiza inferencia lógica, implementando una redefinición del dialecto de reglas de producción del estándar *Rule Interchange Format*, un lenguaje para la unificación de documentos de reglas en diferentes lenguajes. En adición, el motor de inferencia desarrollado usa ontologías escritas en un subconjunto de OWL 2, e implementa una versión del algoritmo Rete para la realización de *matching* en reglas. Este trabajo se engloba en las tecnologías de la web semántica, y pretende ser una alternativa cómoda y fácil de usar que no requiera de traductores entre lenguajes de reglas. También se ha realizado un estudio de rendimiento del motor, en el que se observa el peso que tiene el pobre rendimiento de Python sobre el motor y la influencia del tamaño de la red Rete sobre el tiempo de ejecución.

Aquest treball de fi de grau consisteix en el desenvolupament d'un sistema en el llenguatge Python que realitza inferència lògica, implementant una redefinició del dialecte de regles de producció de l'estàndard *Rule Interchange Format*, un llenguatge per a la unificació de documents de regles en diferents llenguatges. A més, el motor d'inferència desenvolupat utilitza ontologies escrites en un subconjunt d'OWL 2, i hi implementa una versió de l'algorisme Rete per a la realització de *matching* en regles. Aquest treball s'engloba en les tecnologies de la web semàntica, i pretén ser una alternativa còmoda i fàcil d'utilitzar que no requereixi de traductors entre llenguatges de regles. També s'ha realitzat un estudi de rendiment del motor, en el que s'observa el pes que té el pobre rendiment de Python sobre el motor i la influència de la mida de la xarxa Rete sobre el temps d'execució.

This end-of-degree project consists in the development in the language Python of a system that performs logical inference, implementing a redefinition of the production rule dialect of the Rule Interchange Format (RIF) standard, a language to unify rule documents written in different languages. Furthermore, the developed inference engine uses ontologies written in a subset of OWL 2 and implements a version of the Rete algorithm to perform rule matching. This project is encompassed in the semantic web technologies, and pretends to be an easy and convenient alternative that does not require translators between rule languages. A performance study has also been done, in which the weight of the poor performance of Python over the engine and the influence of the size of the Rete network over the execution time can be observed.

Índice general

1. Introducción **6**
- 1.1. Contexto . . . 6
  - 1.1.1. Conceptos y términos . . . 6
  - 1.1.2. Identificación del problema . . . 7
  - 1.1.3. Actores implicados . . . 8
- 1.2. Justificación . . . 9
  - 1.2.1. Trabajo previo . . . 9
  - 1.2.2. Soluciones ya existentes . . . 10
- 1.3. Alcance . . . 11
  - 1.3.1. Objetivos . . . 11
  - 1.3.2. Riesgos . . . 12
- 1.4. Metodología . . . 13
- 1.5. Leyes y regulaciones . . . 14

2. Planificación **15**
- 2.1. Planificación temporal . . . 15
  - 2.1.1. Lista de tareas . . . 15
  - 2.1.2. Gestión del riesgo . . . 21
- 2.2. Presupuesto . . . 22
  - 2.2.1. Costes de recursos humanos . . . 22
  - 2.2.2. Costes de recursos materiales . . . 24
  - 2.2.3. Costes generales . . . 24
  - 2.2.4. Contingencia . . . 25
  - 2.2.5. Riesgos . . . 25
  - 2.2.6. Presupuesto total . . . 25
  - 2.2.7. Control de gestión . . . 26
- 2.3. Cambios en la metodología . . . 26
- 2.4. Cambios con respecto a la planificación y alcance iniciales . . . 27

3. Informe de sostenibilidad **31**
- 3.1. Autoevaluación . . . 31
- 3.2. Dimensión ambiental . . . 31
- 3.3. Dimensión económica . . . 32
- 3.4. Dimensión social . . . 33

4. Base teórica **34**
4.1. Lógica de primer orden . . . 34
4.2. Sistemas basados en el conocimiento y motores de inferencia . . . 36
4.3. El algoritmo Rete . . . 37
4.4. La web semántica . . . 41
4.5. Tecnologías para la web semántica . . . 41
4.5.1. RDF . . . 42
4.5.2. OWL . . . 44
4.5.3. RIF . . . 46

5. Diseño e implementación del sistema **51**
5.1. Análisis de reglas en RIF . . . 52
5.2. Análisis semántico de reglas en RIF . . . 53
5.3. Importación de ontologías . . . 54
5.4. Manipulación de reglas . . . 55
5.5. Creación de la red Rete . . . 62
5.6. Ejecución . . . 68

6. Estudio del rendimiento **76**
6.1. Metodología . . . 76
6.2. Resultados . . . 77

7. Conclusiones **80**
7.1. Competencias técnicas . . . 81
7.2. Trabajo futuro . . . 82

Índice de figuras

2.1. Diagrama de Gantt de la planificación de este trabajo. 20
2.2. Diagrama de Gantt de la planificación actualizada, desde la entrega de seguimiento. 29

4.1. El algoritmo Rete, visto como una "caja negra". 38
4.2. Red Beta de una red Rete de ejemplo, usando nodos de unión. 39
4.3. Red Alfa de una red Rete de ejemplo. 40
4.4. Representación de una afirmación RDF. 42

5.1. Diagrama UML de las clases del módulo `translator`. 52
5.2. Diagrama UML de las clases del módulo `validator`. 54
5.3. Diagrama UML de las clases referentes a reglas en `preparator`. 56
5.4. Diagrama UML de las clases referentes a la red Rete en `preparator`. 63
5.5. Red Alfa de ejemplo implementada con diccionarios. 65

Índice de tablas

2.1. Recursos materiales necesarios por tarea. 19
2.2. Tabla resumen de las tareas del trabajo. 19
2.3. Sueldos brutos anuales y por hora de los recursos humanos. 22
2.4. Coste de los recursos humanos del trabajo, por grupos de tareas. 23
2.5. Costes materiales del trabajo. 24
2.6. Costes generales del trabajo. 24
2.7. Costes de contingencia del trabajo. 25
2.8. Costes derivados de los posibles imprevistos del trabajo. 25
2.9. Presupuesto total del trabajo. 25
2.10. Coste final de los recursos humanos del trabajo, por grupos de tareas. 30

6.1. Tiempos de ejecución (en segundos) por *test*, motor de inferencia y tamaño de ontología. 78
6.2. Tiempos de ejecución (en segundos) corrigiendo la influencia del rendimiento del lenguaje. 79

Capítulo 1

Introducción

1.1. Contexto

Este trabajo de fin de grado está realizado por Adrián Tormos Llorente, estudiante de la especialidad de Computación de Ingeniería Informática en la Universitat Politècnica de Catalunya. En esta rama se forma a los alumnos en el desarrollo de sistemas informáticos completos, complejos y eficientes, tratando sectores tales como la visualización de gráficos, la inteligencia artificial, la programación lógica, la teoría de la computación, compiladores, o el procesado de información masiva.

De los campos anteriores, este trabajo se centra en la representación del conocimiento y los sistemas basados en reglas. El primero de estos es parte del sector de la inteligencia artificial, y en pocas palabras, se refiere a la representación de información de tal manera que pueda ser interpretada por un sistema informático. El segundo campo, también relacionado con la inteligencia artificial, engloba a aquellos sistemas que manipulan conocimiento mediante el uso de reglas. Por otra parte, también gira en torno al concepto de la web semántica, del que extrae tanto ideas como lenguajes.

1.1.1. Conceptos y términos

En pocas palabras, este trabajo de fin de grado consiste en el desarrollo de un sistema informático (programado en el lenguaje Python) basado en reglas que, mediante la inferencia lógica, genera nuevo conocimiento a partir de una base de información o, en su defecto, un conjunto de hechos. Este tipo de sistemas son llamados motores de inferencia. A continuación, se definen los conceptos más importantes relacionados con los campos anteriores y con el trabajo en sí:

Web semántica El concepto de una representación del contenido de la *World Wide Web* de una manera fácilmente procesable por máquinas, en contraposición al paradigma actual de la red, en que su contenido y conocimiento se muestran en un formato preparado para el entendimiento de los humanos: la web sintáctica [1, p. 4-6]. La iniciativa de la web semántica fue creada por el *World Wide Web Consortium* (W3C), y es importante destacar que no se trata de una nueva red paralela a la actual, sino una evolución de la red actual [2, p. 3].

Representación del conocimiento Según [3, p. 4], el campo de estudio del uso de símbolos formales para representar una serie de proposiciones que un agente (informático o no) cree como ciertas.

Razonamiento Manipulación formal de representaciones del conocimiento para producir nuevas proposiciones lógicas. La **inferencia lógica**, por ejemplo, es un método de razonamiento en que se extraen consecuencias lógicas (que son proposiciones) a partir de otras ciertas proposiciones iniciales dadas como premisas [3, p. 3-4].

Sistema basado en el conocimiento Sistema informático caracterizado por la presencia y el uso de una **base de conocimiento**, una representación del conocimiento que simboliza aquello que cree como cierto, y que es usada para razonar. [3, p. 5-7] Una **ontología**, a su vez, es una definición de conocimiento formada por objetos, propiedades y relaciones entre ellos [3, p. 31-32].

Sistema basado en reglas Aplicación informática construida para usar **sistemas de producción**, un tipo de sistema informático que razona mediante reglas de una cierta forma llamadas **reglas de producción**. Este tipo de reglas tienen la forma `si Condición` (se cumple) `entonces Acción` (se ejecuta) [3, p. 117-119].

Motor de inferencia Sistema informático que usa la inferencia lógica para, dado un conjunto de proposiciones lógicas, las premisas, extraer consecuencias lógicas de ellas.

Notar que de muchos de los conceptos mencionados tanto aquí como en la sección de Objetivos se hablará en detalle en apartados posteriores.

1.1.2. Identificación del problema

La *World Wide Web* ha cambiado el mundo. El funcionamiento de las empresas, la comunicación entre las personas, o incluso la forma en que la gente ve películas es diferente a como era antes de la red. De la misma manera, ha habido un cambio en la percepción general de los ordenadores: de servir para cálculos complejos, a procesar información con propósitos tan diferentes como administrar bases de datos o jugar a videojuegos.

El contenido de la red ha sido presentado tradicionalmente en un formato adecuado al entendimiento humano, siendo usado por ejemplo para buscar información, interactuar con otras personas, o hacer la compra. A menudo se usa los llamados motores de búsqueda, como Google o DuckDuckGo, que recopilan páginas web según tienen o no las palabras clave que el usuario quiere [2, p. 1-2]. Sin embargo, y en general, los mismos motores de búsqueda no interpretan el significado de esos resultados que encuentran para el usuario [1, p. 1].

La web semántica es una iniciativa creada e impulsada por el W3C con el objetivo de desarrollar una serie de tecnologías que permitan representar el conocimiento de una manera fácilmente procesable por sistemas informáticos. Estas tecnologías se clasifican principalmente en dos campos estrechamente relacionados: la representación del conocimiento y la inferencia lógica [4]. Es obvio el por qué del primer campo. Por otra parte, el propósito del segundo en la web semántica es el de la generación de nuevo conocimiento por parte de procesos automatizados. Las reglas definen un mecanismo con el que, mediante la inferencia, manipular la información y generar nuevo conocimiento. A su vez, las reglas también se consideran una forma de información [5].

Las tecnologías para la representación del conocimiento que el W3C ha desarrollado (o cuyo uso recomienda) son, entre otras, *Resource Description Framework* (RDF) y *Web Ontology Language* (OWL) [4]. Para cubrir el campo de las reglas y la inferencia, el W3C ha diseñado *Rule Interchange Format* (RIF), cuyo principal propósito es servir de lenguaje intermediario entre otros sistemas de reglas para posibilitar su intercambio [6].

Este último lenguaje (concretamente uno de sus tres dialectos, el de reglas de producción) es el que nos atañe: aunque su propósito es permitir el intercambio de reglas entre sistemas, ¿se puede realizar inferencia lógica sobre reglas escritas directamente en este? ¿Es cómodo? ¿Potente? ¿Suficientemente expresivo? Este es el tipo de preguntas que, desarrollando un motor de inferencia que usa RIF para razonar, espero poder resolver.

1.1.3. Actores implicados

En general, RIF es un lenguaje dirigido a los usuarios de tecnologías para el desarrollo de sistemas basados en reglas. A su vez, los sistemas basados en reglas suelen ser aquellos que presentan una solución a problemas para los que no hay una solución algorítmica obvia o eficiente, la lógica de los cuales varía con frecuencia (por ejemplo, para gestionar clientes o *stocks* en una fábrica), o que pertenecen a campos en los que existen expertos que pueden proporcionar información pero que no disponen de la capacidad técnica para desarrollar algoritmos [7]. Estos sistemas pueden ser, también, una porción de un proceso mayor.

El W3C ha compilado con los años una serie de casos de uso, [8], para ilustrar diferentes situaciones y propósitos para los que RIF puede ser usado.

Por ejemplo, un caso con dos empresas, A y B. A suministra productos a B, y ambas deben ponerse de acuerdo en las condiciones del mismo suministro, como el precio, plazos, etc. Por ejemplo, B podría proponer la regla *"Si A realiza una entrega 10 o más días tarde, será rechazada por B"*. A y B usan una misma base de conocimiento, fundamental para entenderse, pero motores de inferencia diferentes para su gestión. Para poder negociar, A y B deben poder entenderse en un formato común para evitar ambigüedades y malentendidos: RIF.

En otro caso de uso se habla del uso conjunto de lenguajes de ontologías como OWL y de lenguajes de reglas. En campos como la medicina, biología, o servicios web, un grupo de actores puede necesitar intercambiar información en forma de reglas y ontologías por partes iguales. En este caso, se usarían una o más ontologías OWL para describir datos, y una base de reglas para describir sus propiedades y funcionamiento. Esta base de reglas, si se mantiene en el lenguaje RIF, permite su traducción a otros lenguajes de reglas que los agentes puedan utilizar de manera sencilla.

Estos son ejemplos en los que RIF se usa entre empresas, o entre usuarios. Tiene sentido, sabiendo cuál es el principal propósito del lenguaje. Pero, ¿por qué no ir un paso más allá? Si fuera viable un motor de inferencia en el mismo RIF, ¿cómo cambiaría la situación? ¿Qué ventajas supondría? Si se extendiera el uso de RIF más allá del intercambio de reglas, de la misma forma que OWL está extendido para describir ontologías. Si los usuarios de RIF sustituyeran los diferentes motores de inferencia de que disponen actualmente por uno común que usara RIF, ¿qué cambiaría?

Sin ir más lejos, podrían ahorrarse la doble traducción (Lenguaje 1 → RIF → Lenguaje 2) que ahora mismo deben realizar si quieren intercambiar reglas. Por otro lado, podrían establecer plataformas colaborativas para la creación y manipulación de reglas en tiempo real entre ellos (sería una mejora significativa en el primer caso de uso, donde A y B no dependerían de un intercambio de mensajes que podría retrasar o alargar sus negociaciones innecesariamente), que no tendrían que ser traducidas a ningún otro lenguaje para ser usadas.

Así pues, el desarrollo de este trabajo está dirigido a los actuales usuarios de RIF, que cambiarían sus lenguajes de reglas actuales por RIF, pasando a usar este motor de inferencia y ahorrando así tiempo y recursos. Hay que tener en cuenta, sin embargo, que un motor de inferencia no es nada novedoso, y su desarrollo sin más no es algo destacable. Aun así, en el peor de los casos, este trabajo será una alternativa más que los usuarios mencionados tendrán a su disposición. De todas formas, espero que (con un poco de suerte) pueda ser el primer paso de algo mayor.

1.2. Justificación

En este apartado de la memoria se discuten proyectos anteriores relacionados con este y se habla de soluciones aprovechables ya existentes para ciertas porciones del trabajo a realizar.

1.2.1. Trabajo previo

Antes de entrar en sistemas estrechamente relacionados con este trabajo de fin de grado, es interesante ver cuán atrás remontan los motores de inferencia. Los primeros motores usaban sintaxis basadas en Lisp, pues es fácilmente interpretable y manipulable. Un ejemplo de este tipo de motores (y una de las inspiraciones de este trabajo) es *C Language Integrated Production System* (CLIPS), desarrollado entre los años 1985 y 1996 en el *Johnson Space Center* de la NASA. Se describe como "un lenguaje de programación basado en reglas útil para crear sistemas expertos"[9], que es de las mayores aplicaciones que han tenido los motores de inferencia a lo largo del tiempo. Otro ejemplo es Eclipse, presentado en [10, p. 1], que fue desarrollado en 1990 por Hewlett-Packard. Curiosamente, integra CLIPS y añade funcionalidades sobre él.

Con el éxito del lenguaje OWL, han aparecido toda una serie de sistemas informáticos que, desde diferentes enfoques, razonan de alguna manera con él. Un ejemplo de ello es Jena, que se define como "un *framework* para la creación de aplicaciones para la web semántica", desarrollado por la *Apache Software Foundation*. Jena contiene APIs para trabajar con RDF, OWL y un motor de inferencia propio. [11] Otro ejemplo es Racer, trabajo de Haarslev y Möller. En [12], los creadores explican que tiene soporte para DAML+OIL (otro lenguaje para representar conocimiento) [13], RDF y OWL. Racer es únicamente un motor de inferencia, a diferencia de Jena.

Estos son solo dos ejemplos de un número bastante mayor de sistemas. Sin embargo, lo que tienen en común es que no usan RIF como su lenguaje de reglas: para poder trabajar con reglas en este lenguaje, deben ser traducidas al que cada uno de estos sistemas usa. Así pues, este trabajo ofrece una alternativa en lo que se refiere a qué lenguaje usa para razonar.

1.2.2. Soluciones ya existentes

La realización de este motor de inferencia se puede dividir en varias secciones, que se pueden desarrollar de manera más o menos independiente (que no paralela, pues el desarrollo de algunas depende de que otras existan). Sin ningún orden en particular, se trata del análisis léxico, gramatical y semántico del código en RIF, su paso a *eXtensible Markup Language* (XML) (un formato mucho más manejable para el motor), la preparación de la ejecución (preprocesado para crear las estructuras internas que el motor necesite), la ejecución en sí, y la importación de ontologías. Evidentemente, este trabajo no busca reinventar la rueda, así que en aquellas secciones en que sea posible, se aprovecharán soluciones ya existentes. A continuación, se presentan brevemente estas soluciones.

Para el análisis léxico y gramatical del lenguaje se usará *ANother Tool for Language Recognition* (ANTLR), una herramienta para la creación de analizadores léxicos y sintácticos que genera un programa que permite recorrer árboles de sintaxis abstracta [14].

Para importar ontologías se usará la librería RDFLib, un paquete para Python que permite trabajar fácilmente con el lenguaje RDF [15].

Durante la ejecución del motor de inferencia se aprovecharán algoritmos ya existentes como el algoritmo Rete, base de muchos motores de inferencia actuales, que permite de manera eficiente comprobar si una regla se debe activar con ciertos hechos de la base de conocimiento [3, p. 127-129].

Para el resto de secciones no existe una solución que se pueda adaptar o aprovechar.

1.3. Alcance

En esta sección se habla de cuáles son los objetivos del trabajo, y cuáles son los posibles obstáculos y riesgos que se pueden encontrar a lo largo del desarrollo del proyecto.

1.3.1. Objetivos

Para este trabajo se han identificado tres objetivos genéricos, a partir de los que se han especificado subobjetivos concretos que cumplir.

Aprendizaje sobre nuevos conceptos y tecnologías Parte del conocimiento necesario para realizar este trabajo se aleja de aquello que se puede adquirir en el grado de Ingeniería Informática, en su mayoría por tratarse de conceptos más especializados de lo que se puede cubrir en el mismo. Así pues, es necesaria una búsqueda de información sobre ciertos temas, en primer lugar para tener una visión más amplia y general sobre el marco que engloba al trabajo, y después, para poder entender y desarrollar con éxito las diferentes partes del proyecto. Caben destacar los siguientes dos campos:

- **Sobre las tecnologías de la web semántica** La iniciativa de la web semántica, como se explica en apartados anteriores, es fundamental para este trabajo. Es tan importante entender los motivos de la iniciativa y el contexto en que se origina como las tecnologías que se han derivado de la misma, como los lenguajes RIF o RDF, que se usan en este trabajo.
- **Sobre técnicas para la inferencia lógica** Comprender las técnicas para razonar mediante inferencia lógica de manera eficiente es primordial, dado el tema del trabajo. Algoritmos como Rete van a tener que ser entendidos, implementados, y si es posible, optimizados.

Desarrollo de un motor de inferencia Es evidente que lo más importante en este trabajo es el desarrollo de un motor de inferencia que funcione correctamente y sea lo más completo posible. Este objetivo, muy general, se puede dividir en las siguientes metas:

- **Análisis de código escrito en RIF** Las reglas que el motor de inferencia use estarán escritas en el lenguaje RIF, por lo que es indispensable poder analizar léxica, gramatical y semánticamente el lenguaje. Los dos primeros pasos se realizarán sobre su sintaxis de presentación, y el tercero sobre una estructura XML equivalente.
- **Gestión de bases de conocimiento en RDF** Las bases de conocimiento que usará el motor estarán representadas en el lenguaje RDF, por lo que se deberá poder interpretar y manipular de las formas que sean necesarias para el correcto funcionamiento del motor. Opcionalmente, si el desarrollo avanza sin dificultades, se podría evaluar extender el proyecto de manera que también pudiera interpretar lenguajes más complejos como RDF Schema u OWL.
- **Preprocesado** La fase del preprocesado es la preparación de una ejecución del motor dados un conjunto de reglas y una base de conocimiento. Incluye la creación y rellenado de las estructuras de datos que necesitará el motor para su correcto funcionamiento, por ejemplo para el algoritmo Rete, y la manipulación de reglas para transformarlas en otras lógicamente equivalentes pero más eficientes para el motor (por ejemplo, transformando varios cuantificadores existenciales anidados en uno solo). El segundo proceso podría, como alternativa, modificar un archivo de reglas directamente.

- **Gestión y manipulación de la base de hechos** Es muy importante la correcta gestión de la base de hechos, la información que el motor considera que conoce en un cierto momento de la ejecución. Tanto la adición y eliminación de hechos cuando es necesario como los efectos que tiene sobre las reglas que se pueden ejecutar.
- **Implementación de un algoritmo de *pattern matching* en reglas** Se implementará el algoritmo Rete, el más usado para ello. Si se puede, se buscará optimizarlo.
- **Implementación de una estrategia de resolución de conflictos** Se produce un conflicto cuando existen a la vez varias reglas que se pueden ejecutar y hay que escoger un orden para llevarlas a cabo [16]. Es necesaria como poco una estrategia para determinar este orden, y de manera opcional se puede incluir la opción de escoger entre varias.
- **Desarrollo de una interfaz gráfica** En un principio, el motor va a funcionar con una interfaz simple en consola. Si el proyecto se desarrolla sin problemas, se programará una interfaz gráfica, mejorando así la usabilidad del sistema.

Corrección El motor de inferencia debe funcionar sin fallos, por lo que es necesario comprobar tanto teóricamente como mediante una fase de *testing* que todo se ejecuta correctamente, de la manera en que debe hacerlo.

- **Corrección de cada una de las partes del programa** Se obvia la explicación de los subobjetivos de este punto por cuestiones de brevedad. En pocas palabras, la corrección de cada sección del programa es un subobjetivo de esta meta genérica.

Notar que de muchos de los conceptos mencionados tanto aquí como en la sección de Conceptos y términos se hablará en detalle en apartados posteriores.

1.3.2. Riesgos

Los siguientes son los posibles riesgos y obstáculos que se pueden encontrar durante el desarrollo de este trabajo:

Dificultades en la búsqueda de información Como se ha podido comprobar en el apartado anterior, para el correcto desarrollo de este trabajo es necesaria la investigación sobre un buen número de nuevos conceptos. Por una parte, puede ser difícil encontrar información útil, sobre todo en lo que refiere a técnicas de inferencia lógica y razonamiento. Por otra parte, también pueden ser difíciles la asimilación y el entendimiento de los mismos. Por ejemplo, los detalles y complejidades de los lenguajes RIF y RDF pueden ser difíciles de comprender sin experiencia previa con lenguajes similares.

Dificultades en el desarrollo del sistema La creación de un motor de inferencia se divide en un número importante de secciones o tareas. Para la programación de cada una de ellas, es necesario primero el entendimiento de una serie de conceptos avanzados (esto puede ser un problema en sí, ver el punto anterior), y después tener la capacidad de adaptarlos y usarlos correctamente en la programación del motor. Por poner un ejemplo, para la parte del *pattern matching* con reglas se usará el algoritmo Rete. Será importante encontrar información sobre el mismo, entenderla rápidamente y tener la capacidad de programarlo sin problemas. Este ejemplo concreto es, además, crucial, pues se trata de uno de los elementos más importantes del motor.

Problemas de rendimiento Durante la fase de planificación es muy difícil predecir cuál va a ser el rendimiento del motor. Si alguno de los procesos del mismo tuviera un rendimiento poco razonable (ya sea en tiempo de ejecución o en cantidad de memoria usada), habría que invertir más tiempo del esperado en su optimización.

Condiciones externas Siempre hay que tener en cuenta que pueden existir situaciones inesperadas en el futuro que ralenticen el desarrollo del trabajo. Sin ir más lejos, teniendo en cuenta que este trabajo de fin de grado se desarrolla a la vez que se cursan otras asignaturas, siempre se puede dar el caso de que estas requieran más tiempo del que originalmente se creía, haciendo más difícil invertir en el trabajo la cantidad de tiempo recomendada.

1.4. Metodología

Para este trabajo de fin de grado, se usará el proceso de desarrollo iterativo e incremental para la programación del motor de inferencia.

En este tipo de procesos, se divide el desarrollo del producto en una serie de periodos de una cierta duración fija llamados iteraciones. El producto a desarrollar se divide en subconjuntos de funcionalidades, y en cada una de las iteraciones se trabaja en uno de ellos. De esta manera, tras cada una de las iteraciones, se obtiene una versión funcional del producto a desarrollar que, aunque no dispone de todas las funcionalidades y características que el producto deberá tener, sirve de base para seguir añadiendo funcionalidades en la siguiente iteración [17].

En cada una de las iteraciones se pasa por todas las fases del desarrollo de *software* tradicional: se inicia por el análisis de requisitos y el diseño, se pasa a la implementación, y por último se verifica el correcto funcionamiento de la versión [18].

Esta metodología también permite desarrollar varias iteraciones de manera paralela siempre que no existan dependencias entre funcionalidades de diferentes iteraciones [17], pero esto no se usará en este trabajo.

Se ha considerado adoptar también alguna metodología ágil, dado que estas se basan en el desarrollo iterativo de *software*, pero se ha decidido no hacerlo puesto que este tipo de metodologías está más enfocado al trabajo en equipo y este es un trabajo individual [19].

En lo que atañe al seguimiento del desarrollo, se recurrirá a la propia organización personal y la creación de un proyecto en la herramienta de control de versiones Git. Se usarán herramientas como el diagrama de Gantt (Figura 2.1) diseñado, y técnicas como el *bullet journaling* para tener en mente cuál el progreso dentro del conjunto global del trabajo.

Git es un *software* de control de versiones de proyectos con gran énfasis en el soporte a desarrollos no lineales (por ejemplo, con las herramientas para ramificaciones), gestión distribuida (cada desarrollador tiene una copia local del proyecto sobre la que trabaja), visualización del historial del proyecto, y otras características [20]. Esta herramienta permitirá poder comprobar y validar la fecha en las que cierto objetivo se cumple, permitiendo el seguimiento del desarrollo.

Si se diera el caso de surgir imprevistos y verse afectado por alguno de los obstáculos descritos en la sección de Riesgos, se actuaría de la manera descrita en la sección de Gestión del riesgo, más adelante. Se hablará con el director de manera periódica para controlar el progreso, y si fuera necesario en este caso, también para llegar a una hoja de ruta alternativa.

1.5. Leyes y regulaciones

No se ha encontrado ninguna ley ni regulación que esté relacionada con motores de inferencia o bases de conocimiento, o que sea en general relevante para este trabajo.

Capítulo 2

Planificación

2.1. Planificación temporal

La realización de este trabajo abarca varios meses: el trabajo en sí comienza la última semana de agosto de 2019 con el primer acercamiento y estudio del tema del trabajo, y está planeado que termine a finales de enero de 2020 con la lectura del trabajo, que se producirá entre los días 23 y 29 de enero.

Se ha estimado que este proceso durará unas 611 horas (el equivalente a unos 25.5 días) repartidas en aproximadamente 150 días (unos 5 meses). Para comprender mejor la planificación a continuación es importante notar que se dedicarán al trabajo una media de entre 5 y 6 horas diarias. Por supuesto, el tiempo dedicado en un día concreto dependerá de factores como disponibilidad o la carga de trabajo de otras asignaturas que se cursan a la vez.

2.1.1. Lista de tareas

A continuación se listan las tareas identificadas, agrupadas según la sección del trabajo al que pertenecen. Se obvian tareas de "Documentación" o "Validación", pues cada sección del proyecto excepto la de escritura de la memoria escrita las tiene y es evidente que consisten en, respectivamente, documentar el código realizado en esa sección y comprobar que el funcionamiento de la misma es correcto de manera exhaustiva.

Escritura de la memoria escrita

- **Aprendizaje general (10h)** Estudio de los conceptos que rodean y engloban al proyecto de manera general, antes de profundizar en cada uno de ellos cuando toca.
- **Contexto y alcance (15h)** Escritura de la sección correspondiente al contexto y al alcance del trabajo.
- **Planificación temporal (10h)** Realización de la planificación temporal del trabajo, y escritura de la sección correspondiente.

- **Presupuesto y sostenibilidad (15h)** Estudio de los recursos necesarios para el trabajo, su sostenibilidad, y escritura de la sección correspondiente.
- **Documento final (GEP) (15h)** Preparación del documento final de la fase inicial y corrección de las secciones anteriores.
- **Base teórica (20h)** Escritura del conocimiento teórico necesario para entender la memoria escrita. Esta sección puede ser extendida a lo largo del desarrollo si es necesario.
- **Análisis de reglas en RIF (15h)** Escritura de la sección referente a la programación del análisis de código escrito en RIF.
- **Importado de ontologías en RDF (7h)** Escritura de la sección referente al importado de bases de conocimiento en formato RDF.
- **Manipulación de reglas (5h)** Escritura de la sección correspondiente a la programación de las manipulaciones de reglas efectuadas por el motor.
- **Estructuras de datos (10h)** Escritura de la sección que trata las estructuras de datos internas que usa el motor.
- **Implementación del algoritmo Rete (15h)** Escritura de la sección referente a la programación del algoritmo Rete, y las optimizaciones realizadas sobre el mismo.
- **Flujo de ejecución (20h)** Escritura de la sección correspondiente a la estructura del motor de inferencia y al flujo de la ejecución del mismo.
- **Rendimiento (15h)** Escritura de la sección que trata los diferentes estudios de rendimiento realizados a lo largo del desarrollo del proyecto.
- **Documento de seguimiento (10h)** Preparación del documento de seguimiento, con revisión y limpieza de la memoria existente hasta el momento.
- **Finalización de la memoria (25h)** Ampliación de las secciones anteriores donde se necesite, escritura de las secciones finales de la memoria escrita, y revisión y limpieza del documento entero.
- **Preparación de la presentación (30h)** Preparación de la lectura del trabajo ante tribunal y creación del material de apoyo necesario.

Análisis de reglas en RIF

- **Aprendizaje sobre RIF (15h)** Estudio del lenguaje que se interpretará, en que las reglas están construidas.
- **Descripción del lenguaje (14h)** Descripción del léxico y sintaxis del lenguaje mediante una gramática incontextual.
- **Traducción a XML (7h)** Programación de la traducción de reglas en RIF a estructura XML, para su posterior tratamiento.

- **Análisis semántico (16h)** Programación del análisis de la corrección semántica de las reglas en XML.

Gestión de bases de conocimiento

- **Aprendizaje sobre RDF (5h)** Estudio del lenguaje que usará el motor para representar ontologías.
- **Importado de ontologías (RDF) (7h)** Programación del importado de ontologías escritas en RDF para su posterior uso.
- **Aprendizaje sobre OWL (10h)** Estudio de OWL, un lenguaje más potente que RDF para representar ontologías, que pasará a ser el usado tras tener una versión con los elementos fundamentales del motor funcionando correctamente.
- **Importado de ontologías (OWL) (7h)** Análogamente a la tarea de importado anterior, ahora con ontologías escritas en RDF.
- **Adaptación del motor a OWL (20h)** Adaptación del motor a las nuevas características y capacidades que pasan a ser posibles con conocimiento representado en OWL.

Preprocesado y preparación

- **Manipulación de reglas (6h)** Programación del tratado de reglas necesario para ser fácilmente procesables por el motor.
- **Aprendizaje sobre el algoritmo Rete (10h)** Estudio en profundidad del algoritmo Rete, para *pattern matching*, y cómo implementarlo.
- **Creación de la red de Rete (15h)** Programación del proceso para crear la red Rete, una estructura de datos necesaria para el funcionamiento del algoritmo homónimo, en base a la lista de tareas con las que el motor trabajará.
- **Estructuras para bloques de acciones (10h)** Programación de las estructuras de datos necesarias para representar las acciones que una regla puede llevar a cabo (añadir o eliminar hechos de la base de hechos, por ejemplo).

Ejecución del motor

- **Aprendizaje sobre motores de inferencia (20h)** Estudio en profundidad de la estructura de los motores de inferencia y del flujo general de su funcionamiento.
- **Flujo general de la ejecución (15h)** Programación del esqueleto del funcionamiento del motor en sí.
- **Implementación del algoritmo Rete (ejec.) (18h)** Programación del resto del algoritmo Rete, la porción que se lleva a cabo durante el funcionamiento del mismo motor.
- **Resolución de conflictos (6h)** Programación de la parte referente a la resolución de conflictos, y la estrategia de resolución de conflictos necesaria.

- **Otras estrategias de resolución de conflictos (4h)** Programación de estrategias alternativas de resolución de conflictos.
- **Estudio del rendimiento (10h)** Una serie de pruebas para comprobar cuál es el rendimiento del motor.
- **Optimizaciones (15h)** Modificaciones al motor de inferencia para mejorar su rendimiento.
- **Optimizaciones con paralelismo (15h)** Modificaciones al motor, usando el paralelismo, para mejorar mejorar su rendimiento.
- **Estudio del rendimiento (10h)** Análogamente al otro estudio, pero con la nueva versión del motor.

Interfaz gráfica

- **Aprendizaje sobre interfaces en Python (8h)** Estudio del funcionamiento de las interfaces gráficas y cómo programarlas en Python.
- **Interfaz gráfica (25h)** Desarrollo de una interfaz gráfica para el motor.

Tal como se describe en el diagrama de Gantt (Figura 2.1) más abajo, se inicia el desarrollo por el análisis de reglas y el importado de ontologías. Tras ello, se completa el preprocesado y se trabaja en la ejecución del motor para obtener una primera versión con todo lo necesario para funcionar. A la vez, se va trabajando en las secciones correspondientes de la memoria escrita, de manera que este documento nunca se deje de lado. El objetivo de esta planificación es poder obtener una versión funcional del motor antes de pasar al desarrollo de las extensiones y las partes más opcionales: el importado de ontologías en OWL, las optimizaciones y la interfaz gráfica, en este orden.

Este es un trabajo en el que se desarrolla un *software*. Son necesarios, evidentemente, recursos físicos como un PC de suficiente potencia o herramientas de programación. En cuanto a lo último, para la totalidad de las tareas de programación se usará el editor de texto Atom. Para validar la corrección del programa se usará *unit testing* con el paquete de Python *unittest* [21]. También se usará Gitlab en lo que se refiere a seguimiento del desarrollo (ver Metodología). Por otra parte, para las tareas de aprendizaje, se recurrirá a bibliografía tanto en formato físico como digital: libros, *papers*, revistas, noticias (físicas y digitales), documentación de *software*, especificaciones, y otros tipos.

En la Tabla 2.1 se listan los recursos materiales para el desarrollo de cada tarea. Se obvian aquellas tareas, en aras de la brevedad, donde los únicos recursos materiales necesarios son un ordenador y Atom.

En cuanto a recursos humanos, la totalidad se realizará por el autor del trabajo de fin de grado, desepeñando diferentes roles como jefe de proyecto, diseñador, desarrollador o ingeniero de *testing*. Así pues, el autor (con la guía del director del trabajo) es el único recurso humano necesario. Más detalle en la sección de Presupuesto.

En la Tabla 2.2 se puede observar un resumen de la planificación temporal del trabajo. Se agrupan y resumen las tareas para ser breves.

Tareas	Recurso
Aprendizaje general Base teórica Aprendizaje sobre RIF, RDF y OWL Aprendizaje sobre el algoritmo Rete Aprendizaje sobre motores de inferencia Aprendizaje sobre interfaces en Python	Bibliografía relacionada con el tema a estudiar
Tareas de validación	*unittest*

Tabla 2.1: Recursos materiales necesarios por tarea.
Elaboración propia.

ID	Descripción	Horas	Dependencias
1.1	Documentación inicial	65	-
1.2	Secciones del programa	107	1.1, 2.3, 3.3, 4.3, 5.3
1.3	Documento de seguimiento	10	1.2
1.4	Finalización de la memoria	25	1.3, 3.6, 5.5, 6.3
1.5	Preparación de la lectura	30	1.4
2.1	Aprendizaje: RIF	15	-
2.2	Análisis de reglas en RIF	37	2.1
2.3	Documentación y validación	11	2.2
3.1	Aprendizaje: RDF	5	-
3.2	Importado de ontologías (RDF)	7	3.1
3.3	Documentación y validación	8	3.2
3.4	Aprendizaje: OWL	10	-
3.5	Importado (OWL) y adaptación	27	3.4
3.6	Documentación y validación	14	3.5
4.1	Aprendizaje: Rete	10	-
4.2	Preprocesado y preparación	31	4.1
4.3	Documentación y validación	13	4.2
5.1	Aprendizaje: motores	20	-
5.2	Ejecución del motor	43	5.1
5.3	Docum., valid. y rend.	28	5.2
5.4	Optimizaciones	30	5.3
5.5	Docum., valid. y rend.	26	5.4
6.1	Aprendizaje: interfaces	8	5.5
6.2	Interfaz gráfica	25	6.1
6.3	Documentación	6	6.2
TOTAL:		611h	

Tabla 2.2: Tabla resumen de las tareas del trabajo.
Elaboración propia.

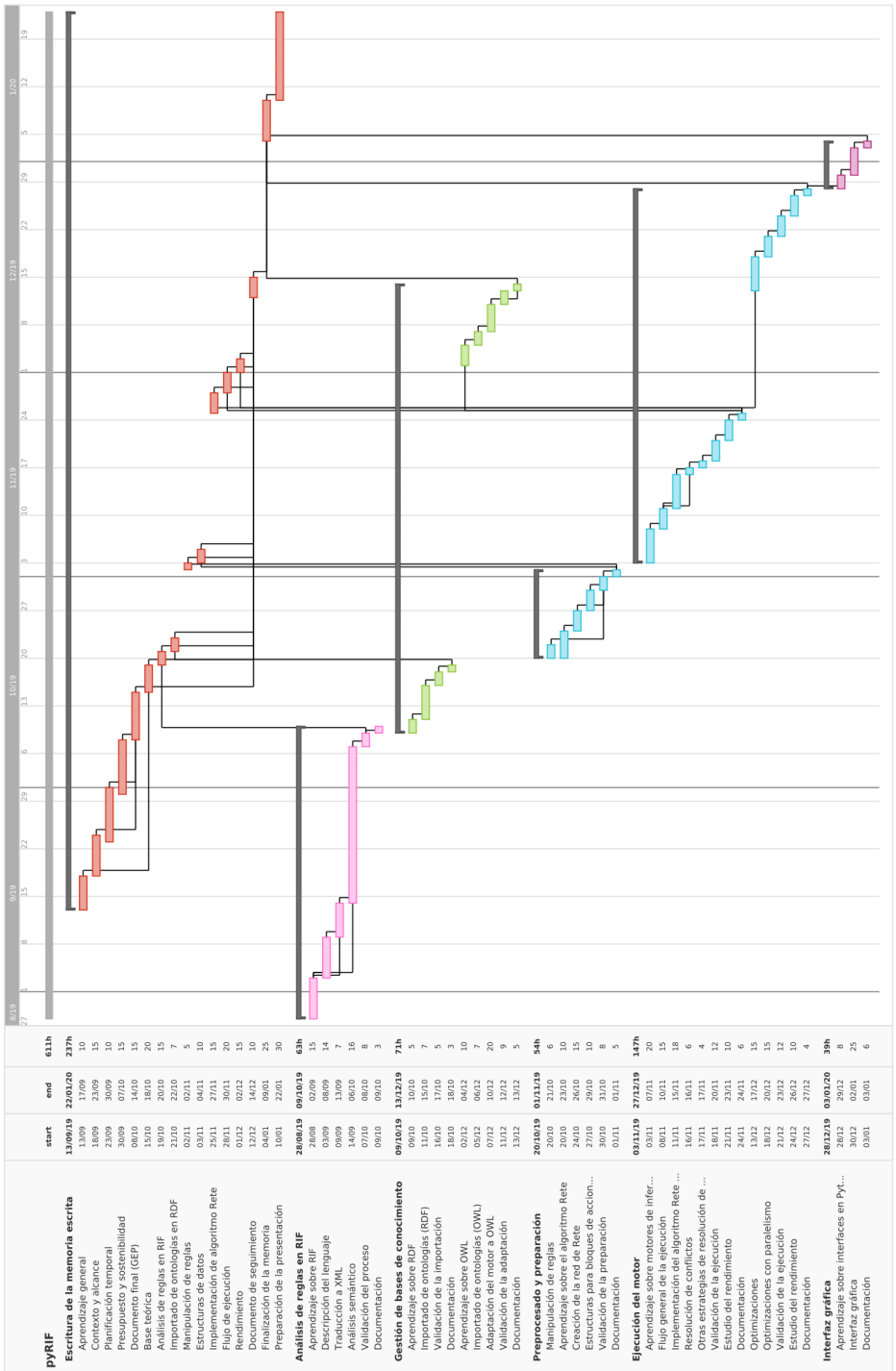

Figura 2.1: Diagrama de Gantt de la planificación de este trabajo.
Elaborado con [22].

2.1.2. Gestión del riesgo

En esta sección se describen las soluciones a los posibles obstáculos y dificultades que se pueden encontrar durante el trabajo, descritas en el apartado de Riesgos.

Dificultades en la búsqueda de información Si se producen dificultades encontrando la información necesaria para las tareas de aprendizaje, lo más probable es que suceda en las de OWL y el algoritmo Rete.

En el primer caso, OWL, se puede estudiar programar la importación de ontologías con la alternativa RDF Schema, un lenguaje a medio camino entre RDF y OWL [23]. Se limitaría un poco, por tanto, el alcance del trabajo, pero a cambio se reduciría ligeramente la duración de la nueva tarea de aprendizaje resultante (es un lenguaje *a priori* más fácil de entender y menos complejo que OWL), compensando en parte la pérdida de tiempo.

En el segundo caso, al no existir realmente una alternativa viable a una parte tan fundamental del motor como el algoritmo Rete, la única estrategia es la identificación lo más pronta posible del problema, y un cambio en el enfoque de la misma búsqueda de información. Es decir, buscar otro tipo de fuentes de información (en vez de bibliografía, por ejemplo, buscar directamente expertos en la materia). El impacto temporal sería inevitable, se estima que unas 5 horas más de trabajo.

Dificultades en el desarrollo del sistema Si se encuentran problemas durante el desarrollo de una porción del sistema, se evaluarán tanto la importancia de la porción en el conjunto del sistema como la gravedad del problema encontrado.

Si se trata de una porción opcional o poco importante (según lo dicho en la sección de Objetivos), se reducirá a algo más sencillo o se eliminará por completo del desarrollo, dependiendo de la gravedad del problema y el impacto temporal que significará. Si se trata de una parte fundamental, no quedará más remedio que buscar más información o pedir ayuda a expertos como el director del trabajo para reducir al mínimo posible el impacto temporal.

Problemas de rendimiento Para los posibles problemas de rendimiento que surjan existen las tareas de optimización. En caso de existir problemas muy graves en alguna de las partes del programa, las tareas de optimización se centrarán en la misma. Si no, consistirán en buscar optimizaciones de manera más general.

Condiciones externas Para reducir el mínimo el impacto de las condiciones externas, imprevisibles por naturaleza, lo mejor es el trabajo constante y continuado. Adicionalmente, si se acaban tareas antes de la fecha prevista, no esperar a las fechas de inicio de las posteriores a realizar sino comenzar a trabajar en ellas antes de lo previsto. Cuanto antes se vayan acabando las tareas, más impacto temporal se podrá recibir después.

2.2. Presupuesto

En este apartado se analiza el coste económico de la realización de este trabajo de fin de grado. En primer lugar se identifican los diferentes costes del trabajo, y tras ello se estimará el valor de cada uno de ellos. Finalmente, se estima el presupuesto total para realizar el trabajo, y se dan mecanismos para controlar los costes del proyecto.

Estos costes se han separado en las siguientes categorías: costes de recursos humanos, costes materiales y costes generales. Los primeros se refieren a aquellos gastos relacionados con las personas que trabajan en el desarrollo, los segundos a aquellos relativos al *software* y *hardware* necesarios, y los últimos a los más genéricos como de electricidad, desplazamiento u otros recursos.

2.2.1. Costes de recursos humanos

En este desarrollo solo se dispone de una persona que hace, según la tarea a realizar, las veces de jefe de proyecto, desarrollador, diseñador, e ingeniero de *testing*. En los cálculos posteriores se entiende la consultoría del director del trabajo como un servicio gratuito ofrecido por la universidad, y por tanto no se incluye ninguna partida al respecto.

En la Tabla 2.3 se muestran los sueldos brutos anuales y por hora medios en España de cada uno de estos roles, extraídos de [24], y su sueldo final teniendo en cuenta el coste de Seguridad Social (un 35 % adicional). Se asume una jornada laboral anual de 1,764 horas, el máximo por ley [25].

Rol	Sueldo anual (€)	Sueldo por hora (€)	Sueldo con S. S. (€)
Jefe de proyecto	35,746€	20.26€	27.35€
Desarrollador	25,809€	14.63€	19.75€
Diseñador	32,033€	18.16€	24.52€
Ingeniero de *testing*	30,375€	17.22€	23.24€

Tabla 2.3: Sueldos brutos anuales y por hora de los recursos humanos.
Elaboración propia.

Conocidos los sueldos de los diferentes roles, se calculan en la Tabla 2.4 los costes de recursos humanos por grupo de tarea (por cuestiones de brevedad), según los grupos de tareas de la tabla resumen de la planificación temporal del apartado anterior.

ID	Descripción	Roles	Horas	Coste
1.1	Documentación inicial	Jefe de proyecto	65	1,777.75€
1.2	Secciones del programa	Jefe de proyecto	107	2,926.45€
1.3	Documento de seguimiento	Jefe de proyecto	10	273.50€
1.4	Finalización de la memoria	Jefe de proyecto	25	683.75€
1.5	Preparación de la lectura	Jefe de proyecto	30	820.50€
2.1	Aprendizaje: RIF	Desarrollador	15	296.25€
2.2	Análisis de reglas en RIF	Desarrollador	37	730.75€
2.3	Documentación y validación	Ing. de *testing*	8	185.92€
		Desarrollador	3	59.25€
3.1	Aprendizaje: RDF	Desarrollador	5	98.75€
3.2	Importado de ontologías (RDF)	Desarrollador	7	138.25€
3.3	Documentación y validación	Ing. de *testing*	5	116.20€
		Desarrollador	3	59.25€
3.4	Aprendizaje: OWL	Desarrollador	10	197.50€
3.5	Importado (OWL) y adaptación	Desarrollador	27	533.25€
3.6	Documentación y validación	Ing. de *testing*	9	209.16€
		Desarrollador	5	98.75€
4.1	Aprendizaje: Rete	Diseñador	3	73.56€
		Desarrollador	7	138.25€
4.2	Preprocesado y preparación	Diseñador	12	294.24€
		Desarrollador	19	375.25€
4.3	Documentación y validación	Ing. de *testing*	8	185.92€
		Desarrollador	5	98.75€
5.1	Aprendizaje: motores	Diseñador	7	171.64€
		Desarrollador	13	256.75€
5.2	Ejecución del motor	Diseñador	11	269.72€
		Desarrollador	32	632.00€
5.3	Docum., valid. y rend.	Ing. de *testing*	12	278.88€
		Desarrollador	16	316.00€
5.4	Optimizaciones	Desarrollador	30	592.50€
5.5	Docum., valid. y rend.	Ing. de *testing*	12	278.88€
		Desarrollador	14	276.50€
6.1	Aprendizaje: interfaces	Desarrollador	8	158.00€
6.2	Interfaz gráfica	Desarrollador	25	493.75€
6.3	Documentación	Desarrollador	6	118.50€
TOTAL:				14,211.32€

Tabla 2.4: Coste de los recursos humanos del trabajo, por grupos de tareas.
Elaboración propia.

2.2.2. Costes de recursos materiales

Los recursos materiales comprenden el *hardware* y el *software* usados para el desarrollo del trabajo.

El trabajo tiene estimadas 611 horas de duración (ver Planificación temporal para más detalle), y para calcular los costes asociados a los recursos en cuestión, se asume que se usan en la totalidad del mismo. El único hardware a tener en cuenta es un ordenador personal y un monitor. En cuanto a *software*, se usarán Git (con Gitlab), LaTeX (con Overleaf), TeamGantt y Windows 10. Todos menos el último son servicios gratuitos [26].

La Tabla 2.8 muestra los costes estrictamente mayores a 0€ de los recursos materiales, asumiendo una amortización de 4 años para el *hardware*, de 3 años para el *software* y una jornada laboral anual de 1,764 horas, el máximo por ley [25].

Recurso material	Coste (€)	Coste proporcional (€)
Ordenador	1000€	86.59€
Monitor	150€	12.99€
Windows 10	145€	12.56€
TOTAL:		112.14 €

Tabla 2.5: Costes materiales del trabajo.
Elaboración propia.

2.2.3. Costes generales

A nivel de costes generales, dado que el trabajo no se realiza en una empresa y por tanto no existe local del que calcular gastos, solo se contemplan los costes relativos al transporte para las reuniones con el director del trabajo o los viajes para buscar material bibliográfico.

Recurso material	Coste (€)
T-Jove	142€
TOTAL:	142€

Tabla 2.6: Costes generales del trabajo.
Elaboración propia.

2.2.4. Contingencia

Dada la larga duración de un proyecto como el de este trabajo, se asume un sobrecoste general para cubrir obstáculos, problemas imprevistos y desviaciones en la planificación temporal.

Tipo	Coste (€)	Ratio	Contingencia (€)
Recursos humanos	14,211.32€	15 %	2,131.70€
Recursos materiales	112.14€	15 %	16.82€
Generales	142€	15 %	21.30€
TOTAL:			2,169.82€

Tabla 2.7: Costes de contingencia del trabajo.
Elaboración propia.

2.2.5. Riesgos

De producirse todos los imprevistos detallados en la sección de riesgos, se estima que el impacto temporal sería de entre unas 10 y 12 horas, todas ellas con rol de desarrollador. Los cálculos se realizan con el valor de 12 horas, para ponerse en la peor situación.

Descripción	Roles	Horas	Coste (€)
Retrasos en el desarrollo	Desarrollador	12	237€
TOTAL:			237€

Tabla 2.8: Costes derivados de los posibles imprevistos del trabajo.
Elaboración propia.

2.2.6. Presupuesto total

Con todos los diferentes costes identificados y estimados, se puede llevar a cabo el cálculo del que debería ser el presupuesto total del trabajo.

Concepto	Coste (€)
Recursos humanos	14,211.32€
Recursos materiales	112.14€
Generales	142.00€
Contingencia	2,169.82€
Riesgos	237.00€
TOTAL:	16,872.28€

Tabla 2.9: Presupuesto total del trabajo.
Elaboración propia.

2.2.7. Control de gestión

Para llevar un control de los costes del proyecto, se llevará al día una tabla de horas dedicadas a cada una de las tareas especificadas en la sección de costes de recursos humanos por rol, y al final de cada fase del trabajo de fin de grado se calculará el coste hasta ese momento.

Si se diera el caso de que la variación en los costes supera el 15 % (de manera que el fondo de contingencia no es capaz de cubrirlo enteramente), se realizará un estudio para identificar en qué momento se han producido los desvíos respecto a la previsión, usando la tabla anterior si se trata de una variación provocada por la cantidad de horas dedicadas a ciertas tareas.

Para calcular estas desviaciones, para cada una de las diferentes partidas por separado, se usará el indicador numérico siguiente:

$$d = \frac{100\,(c_{real} - c_{est})}{c_{est}}$$

Donde c_{real} es el coste real de la partida, calculada a partir del coste de cada recurso y el tiempo que ha sido usado, y c_{est} es el coste estimado para esa misma partida. El valor resultante del cálculo para una partida, d, corresponde al porcentaje de desviación de la misma, por lo que si supera el 15 % se tratará de una variación en los costes de la partida en cuestión que el fondo de contingencia no podrá cubrir.

2.3. Cambios en la metodología

De la metodología que se había previsto usar al inicio de la realización del trabajo a la que se ha llevado a cabo finalmente, se pueden encontrar dos pequeñas diferencias.

En primer lugar, se ha pasado a usar PyCharm, un entorno de desarrollo integrado para el desarrollo de *software* en Python. Este entorno es de pago, pero se ha usado una licencia de estudiante para obtenerlo de manera gratuita.

En segundo lugar, se ha desechado la idea de realizar *unit testing*, pues el desarrollo de la estructura para llevarlo a cabo, dada la gran extensión del trabajo, hubiera necesitado demasiados recursos temporales. De la misma forma, la realización de *unit testing* en sí para cada una de las unidades del sistema también hubiera sido demasiado costosa. Así pues, se ha pasado a realizar *testing* por casos y funcionalidades, definiendo documentos de reglas que cubren diferentes casos (límite o no) y probando que el resultado es el esperado en los diferentes módulos del motor. Estas pruebas se pueden encontrar junto al código fuente del motor.

2.4. Cambios con respecto a la planificación y alcance iniciales

La planificación, a nivel inicial, estima una dedicación temporal media de entre 5 y 6 horas al día. Aunque ha sido posible en una buena cantidad de días alcanzar esa marca, ha habido otros tantos en que no se ha podido. Tampoco ha sido posible recuperar estas pérdidas temporales, con lo que la dedicación ha sido algo menor a la estimada. Por otra parte, una proporción elevada de tareas ha sido realizada en menos horas de las estimadas en dicha planificación. En consecuencia, el desarrollo del proyecto no se ha retrasado tanto como uno podría esperar.

En relación a los riesgos previstos en la sección Riesgos de la Introducción, los únicos que realmente han entorpecido el desarrollo son aquellos relacionados con condiciones externas. Las asignaturas que se cursan a la vez que se desarrollaba el trabajo han necesitado una dedicación que ha quitado, en ciertos momentos, recursos temporales al trabajo. También ha habido ciertas circunstancias personales que han entorpecido el trabajo, pero no es adecuado entrar en detalle sobre ello.

Aunque ha acabado siendo inconsecuente, otro de los riesgos se ha manifestado: el de dificultades en el desarrollo del sistema. Debido a lo complejo del algoritmo Rete, durante el periodo temporal en que se implementó no se había entendido del todo bien, con lo que la implementación existente en el prototipo es subóptima en cuanto a espacio usado. Sin embargo, la mejora de esta implementación se ha realizado junto a otras en la fase de optimización del sistema.

Para minimizar el impacto de estas condiciones se ha seguido lo definido en la sección Gestión del riesgo y se ha trabajado de manera continua en el proyecto.

En el momento temporal de la entrega de seguimiento, generalmente las tareas previstas con la excepción de las referentes a la adaptación del motor al lenguaje OWL se habían realizado con éxito. Hay que exceptuar el primer estudio de rendimiento, que se ha decidido que no era relevante porque el prototipo a estudiar iba a poder ser optimizado de todas formas, y la implementación de estrategias alternativas de resolución de conflictos, que finalmente no han podido ser implementadas.

Hay que tener en consideración también que debido a la extensión del proyecto y la gran cantidad de funciones, funcionalidades y casos que tener en cuenta, realizar una validación exhaustiva de todas ellas llevaría una cantidad de tiempo irrazonable dado el marco temporal de este trabajo. Así pues, se han realizado algunas pruebas para asegurar el correcto funcionamiento de las características básicas, pero es imposible garantizar que el sistema no tenga fallos.

Dados los contratiempos mencionados, se decidió que lo mejor era no realizar las tareas referentes a la interfaz gráfica. Desde el primer momento, se ha tratado esta sección del sistema como opcional y relativamente prescindible, ya que su ausencia no provoca grandes cambios ni en el rendimiento ni en el funcionamiento del sistema. Desde la entrega de seguimiento, se han realizado los grupos de tareas restantes en este orden: integración de OWL al motor, optimizaciones, estudio del rendimiento y finalización del documento escrito. Con la eliminación del desarrollo de una interfaz gráfica, se estimaba que los tiempos de desarrollo del resto de tareas iban a poder mantenerse prácticamente intactos (Figura 2.2). Finalmente, la dedicación a las tareas de optimización ha tenido que ser un poco menor para garantizar la finalización del documento escrito.

Estos pequeños cambios no han supuesto un gran impacto en los objetivos definidos en la sección Objetivos, pues como ya se ha mencionado, el desarrollo de una interfaz gráfica para el sistema siempre ha sido considerado algo opcional. Los objetivos fundamentales no se han visto afectados en gran medida.

En cuanto a costes se refiere, dado que generalmente la mayoría de tareas han sido terminadas en menos tiempo de lo estimado (sea porque el cumplimiento de tarea requería menos tiempo, o porque la dedicación ha tenido que ser menor por restricciones temporales), el coste final del desarrollo del trabajo no supera lo estimado inicialmente. Se puede apreciar en la Tabla 2.10 (en cursiva los costes que han tenido que ser estimados porque la tarea no ha finalizado en el momento de la entrega de este escrito) que el coste en cuanto a recursos humanos del trabajo ha sido menor de lo estimado. Además, se ha añadido a la lista de recursos usados el *software* PyCharm, pero dado que se ha usado con una licencia de estudiante gratuita, no supone ningún sobrecoste.

Por tanto, dado que el resto de conceptos del presupuesto han mantenido su coste estimado, se puede confirmar finalmente que no se ha superado el coste estimado para el proyecto.

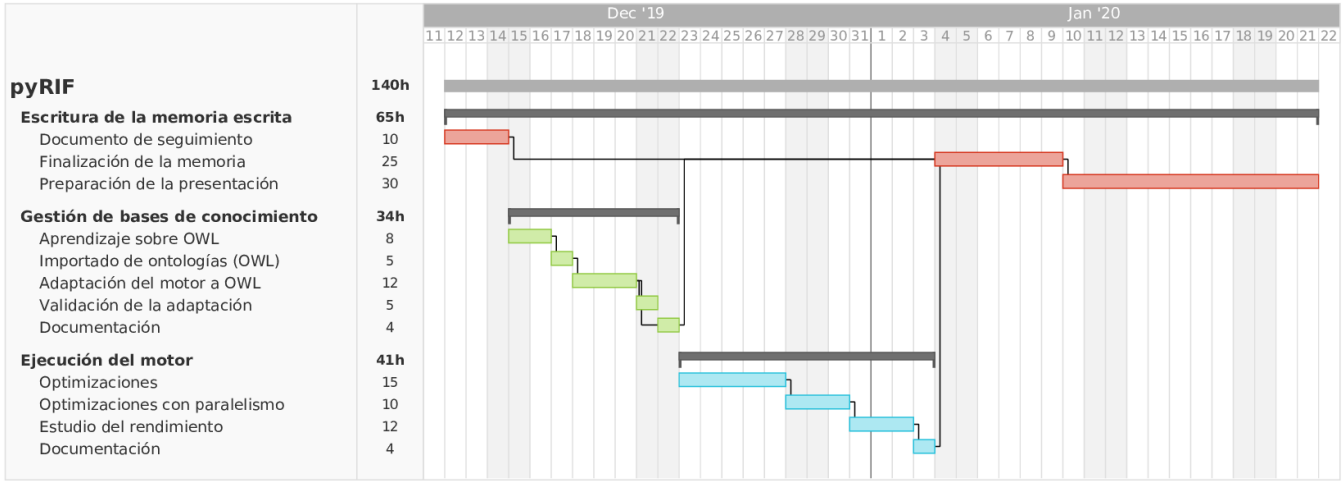

Figura 2.2: Diagrama de Gantt de la planificación actualizada, desde la entrega de seguimiento. Elaborado con [22].

ID	Descripción	Roles	Horas	Coste
1.1	Documentación inicial	Jefe de proyecto	47	1,285.45€
1.2	Secciones del programa	Jefe de proyecto	49	1,340.15€
1.3	Documento de seguimiento	Jefe de proyecto	2.5	68.38€
1.4	Finalización de la memoria	Jefe de proyecto	6	164.10€
1.5	Preparación de la lectura	Jefe de proyecto	*30*	820.50€
2.1	Aprendizaje: RIF	Desarrollador	15	296.25€
2.2	Análisis de reglas en RIF	Desarrollador	36.5	720.88€
2.3	Documentación y validación	Ing. de *testing*	5	116.20€
		Desarrollador	3.5	69.13€
3.1	Aprendizaje: RDF	Desarrollador	2	39.50€
3.2	Importado de ontologías (RDF)	Desarrollador	3.5	69.13€
3.3	Documentación y validación	Ing. de *testing*	1	23.24€
		Desarrollador	1	19.75€
3.4	Aprendizaje: OWL	Desarrollador	5	98.75€
3.5	Importado (OWL) y adaptación	Desarrollador	7.5	148.13€
3.6	Documentación y validación	Ing. de *testing*	2	46.48€
		Desarrollador	1.5	29.63€
4.1	Aprendizaje: Rete	Diseñador	3	73.56€
		Desarrollador	7	138.25€
4.2	Preprocesado y preparación	Diseñador	5	122.6€
		Desarrollador	15.5	306.13€
4.3	Documentación y validación	Ing. de *testing*	6.5	151.06€
		Desarrollador	2	39.50€
5.1	Aprendizaje: motores	Desarrollador	5	98.75€
5.2	Ejecución del motor	Desarrollador	25	493.75€
5.3	Docum., valid. y rend.	Ing. de *testing*	9	209.16€
		Desarrollador	2	39.5€
5.4	Optimizaciones	Desarrollador	13	256.75€
5.5	Docum., valid. y rend.	Ing. de *testing*	12	278.88€
		Desarrollador	1	19.75€
TOTAL:				7,691.92€

Tabla 2.10: Coste final de los recursos humanos del trabajo, por grupos de tareas. Elaboración propia.

Capítulo 3

Informe de sostenibilidad

3.1. Autoevaluación

Las tecnologías de la información y la comunicación, las TIC, se han convertido en algo universal en el mundo moderno. Su llegada ha redefinido por completo el mundo, la sociedad y todo lo que nos rodea, y es innegable que han tenido un impacto enorme en todos y cada uno de nosotros. Es fundamental, entonces, poder comprender cuál es el impacto medioambiental de una tecnología de tal magnitud como esta.

Sin embargo, a título personal, no puedo decir que tenga un gran entendimiento del tema. Aunque pueda conocer cuáles son las técnicas que permiten innovar e idear en el sector, no entiendo en gran medida sobre la huella medioambiental y la sostenibilidad de los proyectos del ámbito de las TIC. Por supuesto, no obstante, es necesario tener interés por aprender sobre ello, y demostrar así la voluntad de aportar y mejorar tanto en el entorno de uno mismo como a la sociedad.

Este trabajo de fin de grado es una puerta para llegar a una mejor comprensión sobre el concepto de sostenibilidad en las TIC. Sin ir más lejos, se discuten los impactos ambientales, económicos y sociales del proyecto en desarrollo, maneras de reducir su huella ambiental... Es importante, así pues, tomar con ganas e interés esta sección del trabajo, pues aporta conocimiento en temas como indicadores de sostenibilidad o cálculos de presupuesto que quedan lejos de aquello que se estudia en este grado, y por ende, puede dar una visión más amplia del desarrollo de proyectos a los alumnos.

Espero, con este trabajo, poder aprender todo lo necesario sobre lo que respecta a la sostenibilidad en las TIC y poder plasmar, a su vez, ese conocimiento tanto aquí como en futuros proyectos.

3.2. Dimensión ambiental

La potencia media de un ordenador de sobremesa cuando está en uso es de 200W, mientras que la de una pantalla LCD es de 25W [27][28].

Para este trabajo, estos son los únicos recursos en cuanto a *hardware* que son necesarios. Asumiendo que se van a usar en las 611 horas que se estima que dura el trabajo (asunción bastante realista, puesto que incluso en las horas dedicadas al aprendizaje este puede o bien ser llevado a cabo con el ordenador, o bien ser este usado para consultas ocasionales), el consumo energético asciende a 137.475kWh $\approx$ 137.5kWh.

Es bien sabido que los ordenadores portátiles tienen un menor consumo energético que los de sobremesa [27], pero lamentablemente, no se dispone de ninguno para la realización de este trabajo. Por otra parte, se intenta reducir la huella ambiental en el uso de material de apoyo (papel, etc.) reutilizándolo siempre que es posible, o usando material reciclado.

Como bien se explica en el apartado de Actores implicados de la sección de Contexto, el lenguaje RIF se usa como intermediario para el intercambio de reglas entre agentes tales como empresas o equipos de investigación. Para poder hacer inferencia, pues, es necesario para ellos traducir reglas escritas en RIF al lenguaje nativo del motor que usen.

Con la introducción de un motor de inferencia que usa nativamente RIF para razonar, se pretende eliminar la necesidad de usar los traductores intermedios para poder reducir la huella ambiental con respecto a las propuestas actuales.

3.3. Dimensión económica

El coste económico de la realización de este trabajo se ha estimado en 16,872.28€, como se puede observar en el apartado de Presupuesto.

De este, los gastos de recursos humanos (14,211.32€) representan la mayor parte, mientras que los gastos materiales y generales son solo una pequeña porción. Todos estos gastos son, de una manera u otra, inevitables, si bien los correspondientes a recursos humanos podrían acabar siendo menores si el trabajo se realiza en menos horas de las que la planificación temporal indica.

También es importante mencionar que este trabajo solo es una pequeña parte de la vida útil de los materiales que se usan, por lo que en ningún caso es necesario sustituirlos y retirarlos tras la conclusión del trabajo.

En lo que concierne a la vida útil del producto, una vez acabado su desarrollo, se vuelve a citar el apartado de Actores implicados de la sección de Contexto. Eliminar la necesidad de usar *software* intermedio para traducir entre lenguajes de reglas implica una posible reducción del coste económico para los agentes que usen RIF. De la misma manera, al ser el motor de inferencia desarrollado aquí *open source*, también se pueden reducir costes si se sustituye un motor de inferencia de pago por este.

3.4. Dimensión social

A nivel individual, este trabajo me debería permitir, en un principio, aprender sobre varios temas que durante el grado que curso no se aprenden en profundidad como motores de inferencia. Además, ya desde el principio me ha permitido conocer la iniciativa de la web semántica y las tecnologías asociadas a esta. Como consecuencia, estoy descubriendo nuevos usos que tiene la red y un nuevo modo de entenderla: como una gran red de información y datos que se podrá usar en el futuro próximo por agentes informáticos.

Tras su finalización, también se espera que este trabajo tenga impacto social, aunque sea a pequeña escala. Como se ha comentado en otros apartados (ver Contexto y Justificación), se espera que este trabajo sea una alternativa a los motores de inferencia actuales, y por tanto sea una nueva opción a disposición de sus usuarios. Por otra parte, también se espera que este motor sea un paso más hacia la web semántica antes mencionada.

Capítulo 4

Base teórica

En esta sección del documento se detalla la teoría necesaria para entender este trabajo de fin de grado, incluyendo aquellos conceptos relevantes para el mismo.

4.1. Lógica de primer orden

La **lógica** ha sido usada para desarrollar sistemas basados en el conocimiento como el mecanismo para representar conocimiento y razonar, proporcionando las herramientas necesarias para realizar inferencia lógica [29, p. 47].

Una lógica se define como el conjunto de una sintaxis, que define qué es una fórmula, y una semántica, que define la interpretación de una fórmula y cuándo se satisface.

En la **lógica de primer orden**, la sintaxis la forman tres tipos de símbolos: las variables, los símbolos de función y los símbolos de predicado. Ninguno de los tres conjuntos de símbolos tiene un tamaño finito. Los símbolos de función y de predicado tienen una aridad, un número mayor o igual a 0 que representa el número de "argumentos"que reciben. Normalmente, se usan las letras $f, g, h...$ para representar símbolos de función, y las letras $p, q, r...$ para los de predicado. Para los símbolos de función de aridad 0, las llamadas constantes, se usan las letras $a, b, c...$ [3, p. 16-17]

Hay tres tipos de expresiones sintácticamente correctas en la lógica de primer orden: los **términos**, los **átomos** y las **fórmulas**.

Todas las variables son términos, y un símbolo de función con el mismo número de términos como "argumentos"que aridad tiene también lo es. Por ejemplo, x o $f(x, y)$ son términos válidos, asumiendo que f tiene aridad 2.

Un símbolo de predicado con los mismos términos como "argumentos"que aridad tiene es un átomo válido. La igualdad entre dos términos también es un átomo válido (siempre que se considere la lógica de primer orden con igualdad; en caso contrario, la igualdad no es parte del lenguaje). Así pues, son ejemplos de átomos $p(f(x, y))$, $p(f(x, y), g(g(y)))$ o $f(x, y) = g(y)$.

Aunque los átomos son una expresión como tal en lógica de primer orden, por sí mismos son fórmulas válidas. A estos, se les puede añadir los conectores de conjunción, disyunción y negación ($\wedge$, $\vee$ y $\neg$ respectivamente) y los cuantificadores universal y existencial ($\forall$ y $\exists$ respectivamente). Un ejemplo de fórmula con estos elementos es $\exists x \forall y \, (p(x, a) \wedge \neg q(y))$.

Una fórmula, o un conjunto de ellas, adquiere un significado cuando se les asigna una **interpretación**. Una interpretación $\mathbb{I}$ es un par $\langle \mathcal{D}, \mathcal{I} \rangle$, donde $\mathcal{D}$ es el dominio, un conjunto cualquiera no vacío de objetos, e $\mathcal{I}$ es un *mapping*.

El *mapping* define las relaciones para los símbolos de función y de predicado. Un símbolo de función, f, de aridad n, será una relación $f : \mathcal{D}^n \longmapsto \mathcal{D}$ mientras que un símbolo de predicado, p, de aridad n, será una relación $f : \mathcal{D}^n \longmapsto \{0, 1\}$. Se define también mu como una asignación para cada variable a un elemento del dominio.

Dado un átomo $\alpha = p(t_1, t_2, ..., t_n)$ (t_i son términos), una interpretación $\mathbb{I}$ y una asignación μ, $\mathcal{I}$ **satisface** α si según $\mathbb{I}$, el resultado de p con esos términos por "argumentos" (una vez sustituidos los símbolos de función por los valores del dominio correspondientes según la misma $\mathbb{I}$) es 1. Además, dado un átomo $t_1 = t_2$ (t_i términos), $\mathbb{I}$ satisface α si según $\mathbb{I}$, los términos t_1 y t_2 tienen como valor el mismo elemento del dominio. Se puede notar como $\mathbb{I}, \mu \models \alpha$. Se puede obviar μ en la notación.

Sean α y β fórmulas cualesquiera, se definen las siguientes reglas cuando aparecen conectores o cuantificadores:

- $\mathbb{I} \models \neg\alpha$ ssi no sucede que $\mathbb{I} \models \alpha$.
- $\mathbb{I} \models \alpha \wedge \beta$ ssi $\mathbb{I} \models \alpha$ y $\mathbb{I} \models \beta$.
- $\mathbb{I} \models \alpha \vee \beta$ ssi $\mathbb{I} \models \alpha$ o $\mathbb{I} \models \beta$.
- $\mathbb{I} \models \exists x, \alpha$ ssi independientemente de la asignación μ, existe alguna asignación μ' para x para la que sucede $\mathbb{I} \models \alpha$.
- $\mathbb{I} \models \forall x, \alpha$ ssi independientemente de la asignación μ, todas las asignaciones μ' posibles para x cumplen que $\mathbb{I} \models \alpha$.

Dadas dos fórmulas $\mathcal{F}$ y $\mathcal{G}$, se dice que $\mathcal{G}$ es **consecuencia lógica** de $\mathcal{F}$ si y solo si para cada interpretación $\mathbb{I}$ que cumple $\mathbb{I} \models \mathcal{F}$, también se cumple que $\mathbb{I} \models \mathcal{G}$.

Por último, se dice que una fórmula está en **forma normal disyuntiva** (DNF, *disjunctive normal form*) cuando tiene la forma de una disyunción de conjunciones: $\alpha_1 \vee ... \vee \alpha_n$, de manera que cada α_i tiene la forma $t_1 \wedge ...t_n$, y los t_i son términos o fórmulas atómicas [30, p. 30].

4.2. Sistemas basados en el conocimiento y motores de inferencia

En general, se llama **sistema basado en el conocimiento** (o sistema experto) a un sistema informático que usa conocimiento sobre cierto dominio para dar solución a un problema de ese mismo dominio como un experto sobre el dominio lo hubiera hecho [29, p. 21].

Si se sigue de manera estricta la definición, se puede llegar a creer que la mayoría de soluciones puramente algorítmicas son también sistemas basados en el conocimiento, pero sin embargo estos nunca han sido considerados como tales. Hay tres características que definen a estos sistemas [29, p. 22], que los distinguen de soluciones algorítmicas:

- La separación del conocimiento y el sistema en sí. El código del sistema, que define el funcionamiento del mismo, no se mezcla con el conocimiento sobre el dominio del problema, sino que este se encuentra separado en forma de base de conocimiento.
- El uso de conocimiento de dominios muy específicos y especializados.
- El uso del conocimiento de manera heurística en vez de algorítmica, debido a querer imitar la manera humana de resolver problemas.

El uso de sistemas expertos presenta múltiples ventajas respecto a sistemas puramente algorítmicos, como por ejemplo la capacidad de distrbución de conocimiento específico (facilidad de distribución de un sistema basado en el conocimiento respecto a un experto humano), facilidad de edición del sistema (gracias a la separación del conocimiento y el sistema), consistencia en las soluciones (en cambio, diferentes expertos humanos podrían dar respuestas diferentes a un mismo problema), entre otras [29, p. 26-28].

También presentan desventajas, como por ejemplo que su conocimiento está limitado al dominio definido, y por tanto podría darse el caso de dar respuestas ilógicas en consecuencia.

Un sistema basado en el conocimiento está formado por dos elementos: la **base de conocimiento** y el **motor de inferencia**.

La base de conocimiento recopila todo la información relevante del dominio del problema. El conocimiento puede ser de naturaleza algorítmica o heurística, según si se trata más de conocimiento factual o de "reglas de oro" que seguir. El formato en que se almacena la base también puede variar: puede almacenarse como predicados, reglas, redes asociativas, *frames* u objetos [29, p. 37-38].

El motor de inferencia es el componente que soluciona problemas del dominio. Usa los contenidos de la base de conocimiento y la información conocida sobre el problema actual para extraer conclusiones e información adicionales. Evidentemente, el motor debe ser compatible con el formato en que está representada la base de conocimiento.

Para llegar a una solución, se puede decir que el motor intenta encontrar conexiones entre los datos que definen el problema y una de las posibles conclusiones. Estas "conexiones"pueden ser creadas desde diferentes enfoques: desde la definición del problema hasta la solución (*forward reasoning*), desde la solución hasta la definición del problema (*backward reasoning*), o desde ambos hasta encontrarse a medio camino (*bidirectional reasoning*) [29, p. 38-39].

Adicionalmente, se llama **sistema de producción** a un sistema que usa *forward reasoning* junto a reglas de una cierta forma llamadas **reglas de producción**. Un sistema de producción mantiene una memoria de trabajo (*working memory*) con aserciones, que puede cambiar durante su ejecución, a medida que se van disparando reglas. Estas reglas, además, siguen la estructura `si Condición` (se cumple) `entonces Acción` (se ejecuta) [3, p. 117-119]. El conjunto de reglas que en un momento dado se pueden ejecutar se llama conjunto de conflicto (*conflict set*), y bajo el supuesto de que no se pueden ejecutar a la vez varias reglas, es necesaria una estrategia de resolución de conflictos para determinar en qué orden se ejecutan estas [31, p. 21].

4.3. El algoritmo Rete

En un sistema de producción, una regla puede ejecutarse cuando su condición se satisface. Para determinar qué reglas pueden ser disparadas dada una memoria de trabajo concreta se usan algoritmos de *matching*. El objetivo de estos es señalar cuáles de las reglas se satisfacen, y para qué elementos de la memoria de trabajo concretos lo hacen.

El algoritmo Rete es un algoritmo para realizar *matching* sobre reglas de manera eficiente desarrollado y publicado por Forgy. Él mismo explica en [31, p. 17] que los algoritmos existentes de *matching* eran poco eficientes cuando se gestionaban muchas elementos y reglas diferentes, llegando a ocupar hasta el 90 % del tiempo total de la ejecución en algunos sistemas.

En pocas palabras, este algoritmo funciona como un módulo que recibe los cambios en la memoria de trabajo y el conjunto de reglas (aunque no todos los motores de inferencia actualizan el conjunto de reglas durante la ejecución), y devuelve como salida los cambios correspondientes en el *conflict set* (Figura 4.1) [32, p. 9].

El algoritmo presenta dos mejoras sobre la tradicional fuerza bruta (tras ejecutar una regla, revisar para cada regla si se cumple para algún conjunto de datos en memoria, probando todas las combinaciones necesarias). La primera, que permite no recorrer tras cada regla disparada todos los elementos en la memoria de trabajo. La segunda, que elimina redundancias entre reglas.

La primera mejora se consigue almacenando información entre ejecuciones de reglas. Para cada elemento de la *working memory*, el algoritmo almacena la lista de condiciones que cumple. Con cada cambio en la memoria de trabajo, la lista se actualiza si es necesario. Cuando se añade un elemento a la memoria, se comprueba qué condiciones de qué reglas cumple, y se genera dicha lista.

Cada uno de estos cambios en la memoria de trabajo se representa como un token:

$$\langle Añadir/Eliminar, Elemento \rangle$$

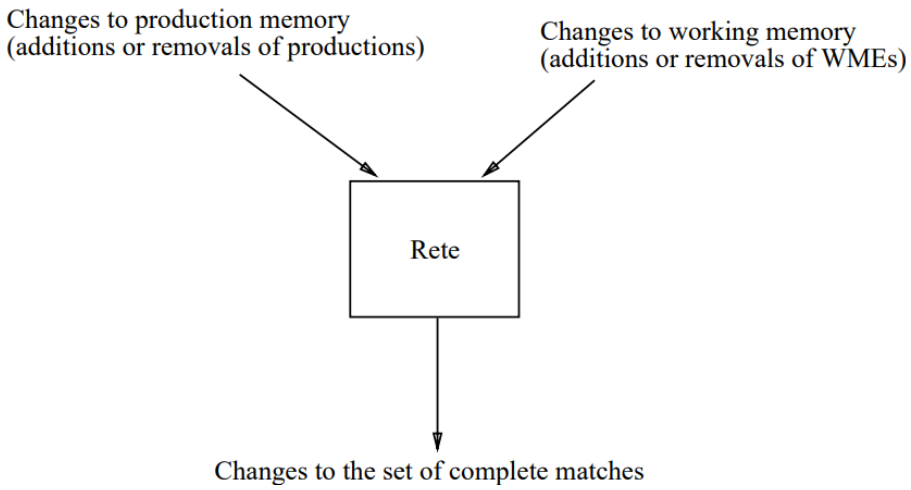

Figura 4.1: El algoritmo Rete, visto como una "caja negra".
Extraído de [32, p. 9].

Para evitar la redundancia en memoria entre reglas cuando estas comparten condiciones, el algoritmo Rete genera una red en forma de árbol para realizar las comprobaciones pertinentes sobre los tokens anteriores. Se podría decir que esta red es el componente principal de la anterior "caja negra".

La red se divide en dos partes, la red Alfa y la red Beta. La red Alfa se encarga de realizar las comprobaciones intra-elemento, es decir, aquellas en que solo interviene un elemento de la memoria de trabajo. La red Beta se encarga de las comprobaciones inter-elemento, aquellas donde intervienen varios elementos de la memoria de trabajo [31, p. 23-24]. Imaginemos que las condiciones tienen la forma $\langle sujeto, propiedad, predicado \rangle$, y que las variables se representan como palabras con el símbolo ? delante. Los siguientes podrían ser ejemplos de comprobaciones intra e inter-elemento, respectivamente:

$$\langle ?persona, nombre, \text{“}manuel\text{”} \rangle$$

$$\langle ?padre, padrede, ?hijo \rangle$$

Cada uno de los nodos de la red Alfa representa una comprobación a realizar sobre un elemento de la *working memory*. Cuando la red recibe un token, lo envía a la red Alfa, donde es distribuido a todos los nodos para determinar cuáles de las comprobaciones supera. Los nodos cuyas comprobaciones supera almacenan entonces dicho token. Los nodos de la red Alfa distribuyen el token, si su comprobación ha sido superada, a los nodos de la red Beta a los que están conectados.

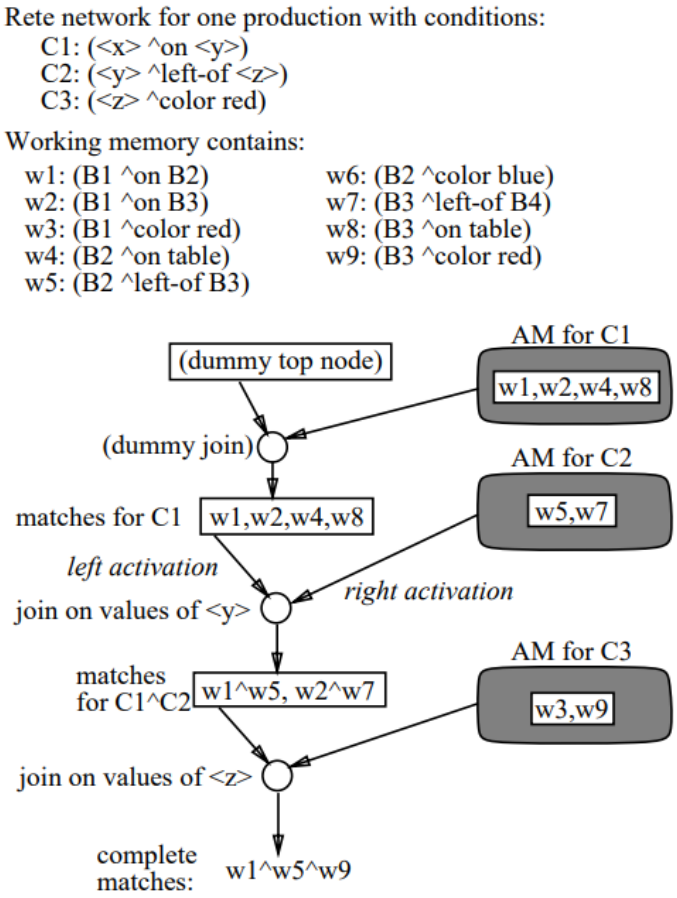

Figura 4.2: Red Beta de una red Rete de ejemplo, usando nodos de unión.
Extraído de [32, p. 10].

Los nodos de la red Beta usan los tokens que reciben para determinar qué combinaciones de elementos de la memoria de trabajo cumplen las condiciones inter-elemento que comprueban. Algunas implementaciones convierten estos nodos en nodos de unión (*Join nodes*), que no comprueban directamente que se cumple la condición si no que unen listas de elementos que cumplen condiciones (Figura 4.2) [32, p.11-13]. Las condiciones inter-elemento pueden, entonces, pasar a ser comprobadas en la red Alfa (Figura 4.3).

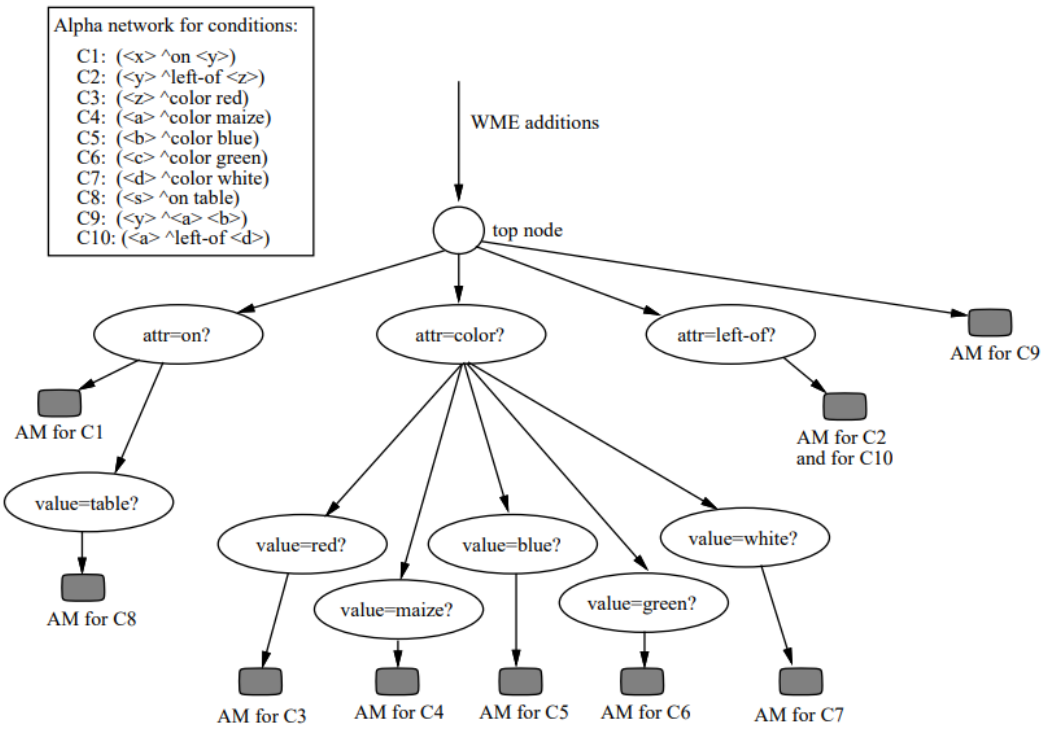

Figura 4.3: Red Alfa de una red Rete de ejemplo.
Extraído de [32, p. 14].

Tras las redes Alfa y Beta se encuentran los nodos objetivo (*Goal nodes* o *Production nodes*), que simbolizan la satisfacción de las condiciones de una regla concreta. A estos, solo llegan aquellos conjuntos de tokens o elementos que cumplen todas las condiciones de la regla, tras haber pasado por las redes Alfa y Beta.

En lenguajes con suficiente expresividad, la condición de una regla puede ser una fórmula booleana cualquiera, con conjunciones y disyunciones. El algoritmo Rete permite expresar condiciones en forma normal disyuntiva, donde cada conjunción de la disyunción es representada en la red por conjuntos de nodos diferentes, que forman caminos diferentes entre el inicio de la red Alfa y el nodo objetivo correspondiente. Por cada conjunción, un camino diferente.

4.4. La web semántica

Se puede llamar **web sintáctica** a la *World Wide Web* (WWW) actual, una red mediante la que se intercambian documentos y archivos multimedia. La red contiene una enorme cantidad de información que se obtiene mediante navegadores web y motores de búsqueda. Cuando un usuario busca información en Internet sobre un cierto tema, usa un motor de búsqueda sobre la WWW para obtener recursos en forma de archivos o documentos digitales, preparados para ser interpretados por usuarios humanos [1, p. 3-4].

Aunque la sintaxis de estos documentos es fácilmente entendible por sistemas informáticos (por ejemplo, qué es un título, qué es una imagen, etc.), no tienen ninguna manera consistente de extraer su significado [33]. Es por eso que aparece la iniciativa de la **web semántica**, impulsada principalmente por Tim Berners-Lee, uno de los creadores originales de la WWW. La web semántica no es una nueva red paralela a la anterior, sino que se trata de un nuevo paradigma de la red en que sus archivos y su información se almacenan en un formato estándar fácilmente procesable por los sistemas informáticos antes mencionados, cuyo significado es por ende entendible para ellos [34]. Se puede ver como una extensión a la red ya existente.

Marshall y Shipman exponen, en [35], tres definiciones diferentes que la web semántica suele tener.

- En primer lugar, como una biblioteca universal, definición que surgió como solución al desorden que tenía la red en la época de su origen. En su momento existía seriamente el temor a acumular en la red una cantidad inmanejable eficientemente de información y documentos, pero el surgimiento de algoritmos de indexado de documentos lo apaciguó, y esta visión de la web semántica ya no es tan relevante.
- En segundo lugar, como una gran base de conocimiento global y distribuida, donde cada documento web tendría una versión preparada para humanos y otra para sistemas informáticos. De esta forma, estos sistemas podrían buscar, filtrar y preparar información para asistir al usuario humano mientras navega [36].
- Por último, como infraestructura para el intercambio coordinado de datos y conocimiento. Según esta visión, desarrolladores crearían bases de conocimiento distribuidas para cierto dominio que usarían aplicaciones del mismo.

4.5. Tecnologías para la web semántica

Para llevar a cabo la extensión a la WWW antes mencionada, el W3C ha diseñado una serie de tecnologías estándar para propósitos tales como la creación de ontologías y reglas para manejar y manipular datos. En este escrito se habla de los lenguajes **RDF**, **OWL** y **RIF**, que son el conjunto de tecnologías relevantes para este trabajo. Los dos primeros son modelos para representar datos, y el último un lenguaje para la representación de reglas [6][37][38].

4.5.1. RDF

El *Resource Description Framework*, **RDF**, es un modelo de representación de información de propósito general. Su principal propósito es representar metadatos (información que describe el contenido de un documento o archivo; en pocas palabras, datos sobre datos) sobre recursos de la red (páginas web, por ejemplo), pero también puede ser usado para expresar información sobre objetos de los que se puede encontrar información en la red. Por ello, puede ser empleado como un lenguaje de ontologías ligero [1, p. 57, 62]. El estándar de este modelo está expuesto en [37].

La base de RDF son los llamados *statements*, que podrían ser traducidos como **afirmaciones**. Una afirmación es una tripla (S, P, O), donde S es el sujeto, P la propiedad y O el predicado (en inglés *object*) de la afirmación. Dada una afirmación (S, P, O), se puede decir que "S tiene una propiedad P con valor O"[1, p. 62]. Cada uno de los elementos anteriores puede ser un *Internationalized Resource Identifier* (**IRI**) o un **literal**.

El IRI es un estándar definido en [39] por Duerst y Suignard para la creación de identificadores de recursos. Un identificador IRI representará cualquier objeto, concepto, persona, etc., del que se quiera expresar información [1, p. 62][37]. Un ejemplo de IRI es:

```
http://www.perceive.net/schemas/relationship/enemyOf
```

Un *namespace* es una colección de identificadores, y también se identifica con un IRI. Estos permiten, en algunas sintaxis de RDF, definir abreviaciones para los IRI. Si se define un **prefijo**, una abreviatura para un *namespace*, este se puede unir al nombre de un recurso, separándolo por dos puntos (":"), para formar un identificador válido. Por ejemplo, dado el prefijo `foaf` para el *namespace* `http://xmlns.com/foaf/0.1/`, la propiedad `http://xmlns.com/foaf/0.1/knows` puede ser abreviada como `foaf:knows`.

Un literal es una cadena de caracteres que representa una constante de un cierto tipo. Los literales solo pueden usarse como predicados de afirmaciones, nunca como sujetos o propiedades. Un literal está formado por dos o tres elementos: una cadena de caracteres que representa el valor de la constante, otra que representa su tipo (en forma de IRI) y opcionalmente, una etiqueta con el idioma asociado a la constante (una etiqueta que sigue el estándar definido en [40]). Un ejemplo de literal es:

```
"1"^^xs:integer
```

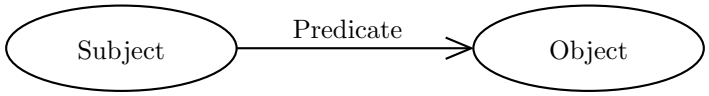

Figura 4.4: Representación de una afirmación RDF.
Extraída de [37].

Un **documento RDF** está formado por un conjunto no ordenado de afirmaciones. Existen varias sintaxis o notaciones para representar documentos RDF, como RDF XML, Turtle, RDFa, JSON-LD o TriG. A continuación hay un ejemplo de documento RDF de tres afirmaciones, extraído de [41].

```
<http://example.org/spiderman>
  <http://www.perceive.net/schemas/relationship/enemyOf>
  <http://example.org/green-goblin>
<http://example.org/spiderman>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person>
<http://example.org/spiderman>
  <http://xmlns.com/foaf/0.1/name>
  "Spiderman"
```

Este se escribiría en notación Turtle [41] como:

```
@base <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .

<#spiderman>
    rel:enemyOf <#green-goblin> ;
    a foaf:Person ;
    foaf:name "Spiderman" .
```

O en notación TriG [42] como:

```
@prefix : <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .

:G1 { :spiderman a foaf:Person ;
              rel:enemyOf :green-goblin ;
              foaf:name "Spiderman" . }
```

La forma de las afirmaciones de RDF, las triplas, permite la fácil transformación de un documento RDF a un grafo. De hecho, un conjunto de afirmaciones, lo que se ha llamado antes un documento RDF, es también llamado **grafo RDF**. Un grafo RDF es dirigido, y puede ser visualizado como un conjunto de nodos (los sujetos y los predicados) unidos por aristas dirigidas. Cada arista une el sujeto de una afirmación con el predicado correspondiente, y contiene como atributo adicional la propiedad que representa.

Más allá de la propiedad `rdf:type`, RDF no presenta ningún medio para definir clases y propiedades específicas para una aplicación. Para ello, existe la extensión **RDF Schema**, con la que se pueden definir ambas y además crear jerarquías. Esta extensión define el espacio de nombres `http://www.w3.org/2000/01/rdf-schema#` [1, p. 71] y, usando el espacio anterior, prové los IRI `rdfs:Class`, `rdfs:subClassOf`, `rdfs:Property`, `rdfs:subPropertyOf`, `rdfs:domain` y `rdfs:range` [1, p. 78].

La primera se usa para definir clases específicas, y la segunda para crear una jerarquía de clases. Las dos siguientes funcionan análogamente para propiedades. Las dos últimas sirven respectivamente para definir el conjunto de clases a que se aplica una propiedad, y los posibles tipos de los valores que una propiedad puede tener.

4.5.2. OWL

El *Web Ontology Language*, **OWL**, es un lenguaje que describe clases, propiedades y relaciones entre conceptos. Es un vocabulario, tal como los lenguajes anteriores, y las ontologías escritas en este tienen la misma estructura que en RDF y RDF Schema. OWL está organizado en tres sublenguajes: OWL Lite, OWL DL y OWL Full, cada uno más expresivo que el anterior. OWL Full se define como una extensión a los lenguajes RDF y RDF Schema, mientras que los otros dos son extensiones a versiones reducidas de los anteriores [1, p. 81]. Se puede encontrar una descripción del lenguaje en [43].

OWL define el *namespace* `http://www.w3.org/2002/07/owl#` y un conjunto de recursos IRI que lo usan.

En primer lugar, define la separación entre propiedades de objetos y de datos. Ls primeras son aquellas propiedades cuyo rango son instancias de objetos, mientras que las segundas las que tienen como rango un conjunto de tipos de datos. Para definirlas, se usan los constructos `owl:ObjectProperty` y `owl:DatatypeProperty`. También se pueden denotar dos o más propiedades como equivalentes con `owl:equivalentProperty`.

También introduce propiedades para las mismas propiedades, llamadas características. Por ejemplo, se puede expresar que una propiedad es transitiva, simétrica, funcional, o que tiene una función inversa (y concretarla). Los IRI para expresar tales características, respectivamente, son `owl:TransitiveProperty`, `owl:SymmetricProperty`, `owl:FunctionalProperty` y `owl:InverseOf`.

Además de las anteriores, añade restricciones de cardinalidad para las propiedades, respecto a las instanciasa definidas. Estas se pueden expresar con `owl:cardinality` (cardinalidad exacta), `owl:minCardinality` y `owl:Cardinality` (cardinalidades mínima y máxima respectivamente).

OWL también expande la expresividad de las jerarquías de clases, añadiendo las clases `owl:Thing` y `owl:Nothing`. La primera representa el universo entero, y todo individuo pertenece a ella. La segunda representa el conjunto vacío, y ningún elemento puede pertenecer a ella.

También permite operaciones de conjuntos con clases, como la unión, la intersección y el complemento, para la expresión de definiciones más complejas en relación a las clases. Se usan los constructos `owl:unionOf`, `owl:intersectionOf` y `owl:complementOf` respectivamente.

Por último, permite definir también clases equivalentes y disjuntas con `owl:equivalentClass` y `owl:disjointWith`.

La última revisión del lenguaje es **OWL 2**, y añade nuevas funcionalidades (tanto azúcar sintáctico como realmente nuevas características que añaden expresividad). Por ejemplo, mayores capacidades para los tipos de datos, más expresividad en las restricciones de cardinalidad o las propiedades asimétrica, reflexiva y disjunta [38].

A continuación se muestra una extensión del ejemplo anterior para RDF con algunas de las propiedades de OWL y OWL 2, en su sintaxis propia OWL/XML [43]:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:per="http://www.perceive.net/schemas/relationship/"
     xmlns:foaf="http://xmlns.com/foaf/0.1/"
     xml:base="http://example.org"
     xmlns="http://example.org#">

  <owl:Ontology rdf:about="http://test.org/onto.owl"/>

  <owl:Class rdf:about="#Person">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
  </owl:Class>

  <owl:ObjectProperty rdf:about="http://xmlns.com/foaf/0.1/enemyOf">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#SymmetricProperty"/>
  </owl:ObjectProperty>

  <owl:DatatypeProperty rdf:about="foaf:name">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>

  <Person rdf:about="#spiderman">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
    <per:enemyOf rdf:resource="green-goblin"/>
    <foaf:name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      Spiderman
    </foaf:name>
  </Person>

</rdf:RDF>
```

4.5.3. RIF

El *Rule Interchange Format*, **RIF**, es un lenguaje de reglas diseñado como un estándar para facilitar la integración de conjuntos de reglas en diferentes lenguajes, formado por un conjunto de dialectos interconectados que representan diferentes características que los lenguajes de reglas pueden tener [44].

El W3C define tres dialectos: el *Core Dialect* (RIF-Core), el *Basic Logic Dialect* (RIF-BLD) y el *Production Rule Dialect* (RIF-PRD). El primero es un núcleo con las características más básicas que comparten todos los dialectos [44], el segundo es un dialecto para razonar con cláusulas de Horn [45] y el tercero, un dialecto para trabajar con reglas de producción [16].

Para este trabajo, el dialecto relevante es PRD. Un documento de este dialecto está formado por una serie de directices `import` para importar documentos, la definición de cero o más prefijos (cuyo funcionamiento es equivalente al de los prefijos en RDF) y cero o un grupos (que contienen las reglas y otros grupos). Los grupos, además de contener reglas y otros grupos, pueden tener asignadas una prioridad y una estrategia de resolución de conflictos. Sin embargo, solo se puede especificar una estrategia de resolución de conflictos por documento.

Una **regla de producción** en PRD está formada por una **condición** y una o más acciones (que forman un **bloque de acciones**). Según el tipo de condición, se definen tres tipos de reglas: bloques de acciones sin condición, bloques de acciones con condición y reglas con definición de variables. Esta es, respectivamente, su sintaxis:

```
Do (ACCIÓN*)
If CONDICIÓN Then Do (ACCIÓN*)
Forall VARIABLE* such that (CONDICIÓN+) (REGLA)
```

Una condición es una fórmula lógica formada por átomos, los conectores de conjunción, disyunción y negación, y el cuantificador existencial. Semánticamente, son iguales a sus equivalentes en lógica de primer orden. La sintaxis para el uso de conectores y cuantificador es, respectivamente, la siguiente:

```
And (CONDICIÓN*)
Or (CONDICIÓN*)
Not (CONDICIÓN)
Exists VARIABLE* (CONDICIÓN)
```

Es importante notar que una condición puede consistir en un solo átomo, sin cuantificadores ni conectores. También, que el W3C obliga por definición a que estas fórmulas estén siempre en forma normal disyuntiva (DNF).

Los tipos de **átomos** que se definen son predicados, expresiones de igualdad, expresiones de pertenencia a una clase, expresiones de pertenencia de una clase a otra, *frames*, y predicados definidos externamente. Tomando que todos los elementos `s`, `t` y `v` son términos, a continuación se define la sintaxis para los diferentes tipos de átomos, respectivamente:

```
t(s1, ..., sn)
s = t
s # t
s ## t
s[t1 -> v1, ..., tn -> vn]
External( t(s1, ..., sn) )
```

El W3C define en [46] una lista de **predicados y funciones estándar** (ambos pueden verse como llamadas a funciones, con la diferencia de que un predicado tiene como valor resultante un booleano y una función no) que usar a la hora de crear reglas en RIF, aunque los usuarios del lenguaje pueden especificar los suyos.

Por último, se definen cuatro tipos de **términos**: constantes, variables, listas y predicados (los predicados se consideran elementos básicos, términos, pero pueden formar fórmulas por ellos mismos). Una lista no puede contener variables.

La sintaxis de una constante es la misma que en el lenguaje RDF; la de una variable, `?NOMBRE` (donde `NOMBRE` es una secuencia de caracteres válida); la de una lista, `List(e1, ..., en)`; y la de los predicados está definida encima.

Por otra parte, se definen cuatro tipos de **acciones** diferentes para los bloques de acciones: las aserciones, las retracciones, las modificaciones y las ejecuciones. Se pueden realizar aserciones de predicados, *frames* o pertenencias a clases; retracciones de predicados, *frames* (también se pueden retractar varios *frames* correspondientes a un objeto), y todas las aserciones a la vez de un objeto; se pueden modificar *frames*; y se pueden ejecutar predicados (externamente definidos o no), entendiéndose como la ejecución de funciones.

La sintaxis de las acciones es la siguiente (todos los `s`, `t` y `o` son términos):

```
Assert (t)
Retract (t)
Retract (o s)
Retract (o)
Modify (t)
Execute (t)
```

Las tres sintaxis diferentes para las retracciones corresponden respectivamente a la retracción de un término simple (predicado, *frame* o membresía), a la de todos los valores para un atributo de un *frame* de un objeto, y a la de todos los hechos en memoria relacionados con un objeto concreto.

Además, los bloques de acciones también permiten definir variables, llamadas **variables de acción** (*action variables*).

El dialecto PRD, como todos los dialectos del W3C, tiene definidas dos sintaxis totalmente equivalentes e intercambiables: una de presentación y otra estándar en XML. En los ejemplos anteriores se ha usado la de presentación, pues es mucho más clara que la XML y se creó con la intención de ser la usada por humanos. A continuación se presenta un ejemplo (extraído y adaptado de [16]) de documento tanto en sintaxis de presentación como en sintaxis XML:

```
Document (
    Prefix( ex1  <http://www.example.com/2009/ex#> )
    Prefix( rif <http://www.w3.org/2007/rif#> )
    Prefix( pred <http://www.w3.org/2007/rif-builtin-predicate#> )

    Group rif:forwardChaining 10 (
        Forall ?customer such that (And( ?customer # ex1:Customer
                                         ?customer[ex1:status -> "Silver"] ))
            (Do (Assert(ex1:silver-customer(?customer))))
    )
)
```

```
<Document>
  <payload>
    <Group>
      <behavior>
        <ConflictResolution>
          <Const type="http://www.w3.org/2007/rif#iri"
            http://www.w3.org/2007/rif#forwardChaining
          </Const>
        </ConflictResolution>
        <Priority>
          <Const type="http://www.w3.org/2001/XMLSchema#integer">10</Const>
        </Priority>
      </behavior>
      <sentence>
        <Forall>
          <declare><Var>?customer</Var></declare>
          <pattern>
            <And>
              <formula>
                <Member>
                  <instance><Var>?customer</Var></instance>
                  <class>
                    <Const type="http://www.w3.org/2007/rif#iri">
                      http://www.example.com/2009/ex#Customer
                    </Const>
                  </class>
                </Member>
              </formula>
              <formula>
                <Frame>
                  <object> <Var>?customer</Var> </object>
                  <slot ordered="yes">
                    <Const type="http://www.w3.org/2007/rif#iri">
                      http://www.example.com/2009/ex#status
                    </Const>
                    <Const type="http://www.w3.org/2001/XMLSchema#string">
                      Silver
                    </Const>
                  </slot>
                </Frame>
              </formula>
            </And>
          </pattern>
          <formula>
            <Do>
              <actions ordered="yes">
```

```
                <Assert>
                  <target>
                    <Atom>
                      <op>
                        <Const type="http://www.w3.org/2007/rif#iri">
                          http://www.example.com/2009/ex#silver-customer
                        </Const>
                      </op>
                      <args ordered="yes"><Var>?customer</Var></args>
                    </Atom>
                  </target>
                </Assert>
              </actions>
            </Do>
          </formula>
        </Forall>
      </sentence>
    </Group>
  </payload>
</Document>
```

Es importante especificar también en este trabajo, por último, que RIF (y por consecuencia PRD) explicita que en una ejecución, una determinada regla solo puede ser disparada una vez para un determinado conjunto de elementos de la *working memory* que satisfacen su condición.

Se puede encontrar una definición más completa y ordenada del lenguaje, junto con una explicación en detalle de la semántica del mismo, en [16].

Capítulo 5

Diseño e implementación del sistema

De los dos principales elementos que forman un sistema basado en el conocimiento, este trabajo se centra en el desarrollo del motor de inferencia.

El sistema en sí, programado en Python, consta de varios módulos: el de traducción de reglas, el de validación de reglas, el de preparación de la ejecución, y el de la ejecución en sí. De cada uno de ellos se habla en detalle en su respectiva sección.

El programa dispone de una simple interfaz de consola, en la que se pueden escribir varios comandos para realizar una de las diferentes funciones del motor de inferencia. A continuación se da una breve explicación de cada uno. Entre paréntesis se escriben los parámetros del comando.

- `translate (source) (dest)` Traduce un documento RIF en sintaxis de presentación, (`source`) a un documento equivalente en sintaxis XML (`dest`).
- `validate (source)`: Comprueba la validez de un documento RIF en sintaxis XML (`source`). Es decir, realiza todas las comprobaciones necesarias para asegurar que cumple todas los requisitos para ser un documento RIF válido y compatible con el motor.
- `prepare (source) (id)`: Genera, para un documento RIF válido en sintaxis XML (`source`), una red Rete que lo representa. Tras ello, la almacena internamente con el identificador `id` para ser usada posteriormente por el motor. Este comando asume que el documento ha sido validado anteriormente.
- `import (source) (id)`: Importa una ontología OWL 2 (`source`) y la almacena internamente con el identificador `id`.
- `execute (ont) (rules)`: El motor de inferencia ejecuta el documento RIF preparado y almacenado como `rules` con la ontología preparada y almacenada como `ont`.
- `status`: Escribe por pantalla una lista con todas las ontologías y documentos RIF almacenados internamente y preparados para ser usados por el motor.

- `exit`: Cierra el programa.

Cuando se quiere escribir el nombre de un archivo como parámetro, se debe tener en cuenta que se necesita incluir la extensión del archivo.

5.1. Análisis de reglas en RIF

El motor de inferencia funciona usando una base de conocimiento escrita en RDF y un documento de reglas escrito en RIF que referencia la base.

El análisis léxico y sintáctico de documentos RIF se ha realizado sobre la sintaxis de presentación con el soporte de la herramienta ANTLR4, que en base a la definición de la gramática de un lenguaje genera un analizador sintáctico (*parser*) para el mismo que genera árboles sintácticos que pueden después ser visitados [14].

El módulo del análisis léxico y sintáctico de reglas, `translator`, está formado por el código generador del analizador sintáctico, y el código visitante de árboles sintácticos. Además, como su nombre indica, este módulo traduce código escrito en la sintaxis de presentación del dialecto RIF-PRD a código equivalente en sintaxis XML.

Excepto `rifActualVisitor`, todas las clases del módulo son autogeneradas por ANTLR4 a partir de la gramática definida para el lenguaje, que se encuentra en el archivo `rif.g4`, de elaboración propia en base a la definición del W3C en [16].

La clase `rifVisitor` es una interfaz que permite la implementación de clases que trabajen con árboles sintácticos, visitándolos nodo a nodo. La implementación para este sistema es la clase `rifActualVisitor`, que dado un documento RIF en sintaxis de presentación, visita nodo a nodo el árbol sintáctico generado por las clases `rifLexer` y `rifParser` para el mismo, y genera un nuevo archivo XML con la traducción del documento a sintaxis XML, de acuerdo con su definición.

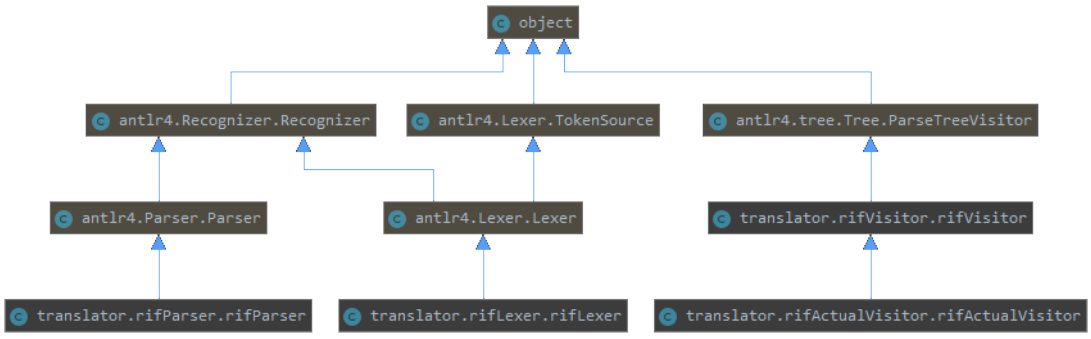

Figura 5.1: Diagrama UML de las clases del módulo `translator`.
Generado con PyCharm y yFiles.

La razón para generar traducciones a sintaxis XML es que el motor de inferencia trabaja internamente con archivos en esta sintaxis, tanto para el análisis semántico como para la preparación de la ejecución y la ejecución propiamente dicha. La decisión de diseño de escoger esta sintaxis por delante de la de presentación viene motivada por la misma definición del lenguaje RIF, en que se insta a trabajar con esta puesto que es la única propiamente normativa.

Es importante notar que dado que el principal propósito de RIF es ser usado como un estándar para integrar reglas en diferentes lenguajes, ha sido necesario realizar algunas pequeñas adaptaciones para usarse para hacer inferencia en este motor. Por ello, se han añadido algunas restricciones y consideraciones adicionales, de las que se habla en el siguiente apartado.

5.2. Análisis semántico de reglas en RIF

El análisis semántico de las reglas corre a cargo del módulo `validator`, y se realiza ya sobre documentos RIF con sintaxis XML.

Antes de entrar en el proceso del análisis en sí, es necesario especificar las diferencias entre el lenguaje RIF-PRD especificado en [16] y el lenguaje que usa el motor, pues como se explica en el apartado anterior, se han añadido algunas restricciones y consideraciones adicionales al funcionamiento del lenguaje.

- Los comentarios se usan para dar un nombre a las estructuras como grupos o reglas, mientras que en el lenguaje original tienen un propósito más amplio, por ejemplo sirviendo para especificar el autor de una regla concreta.
- Se omiten las directivas `import`, pasando a trabajar con un único documento de reglas.
- Los predicados y predicados externamente definidos pasan a representar hechos y llamadas a funciones respectivamente, a diferencia del lenguaje original donde ambos tenían ambos propósitos.
- Cualquier variable que se use en la condición de una regla debe haber sido previamente definida en una regla con definición de variables o en un cuantificador existencial. En el lenguaje original se podían usar variables sin definir previamente si estaban relacionadas de alguna manera con otra que sí está definida (por ejemplo, como valor en un *frame* donde el objeto es una variable ya definida).
- Los predicados externamente definidos no podrán ser parte de otro átomo que no sea a su vez un predicado externamente definido. Además, solo los predicados externamente definidos que resultan en un valor booleano pueden ser un átomo por sí mismos. En el lenguaje original, en cambio, no existía ninguna definición clara al respecto.
- Se han limitado los tipos que soporta el motor a un subconjunto de los definidos para el lenguaje: `xsd:boolean`, `xsd:integer`, `xsd:decimal`, `xsd:double`, `xsd:string` y `rif:iri`. De la misma manera, también se ha implementado solo un subconjunto estricto de las funciones definidas.

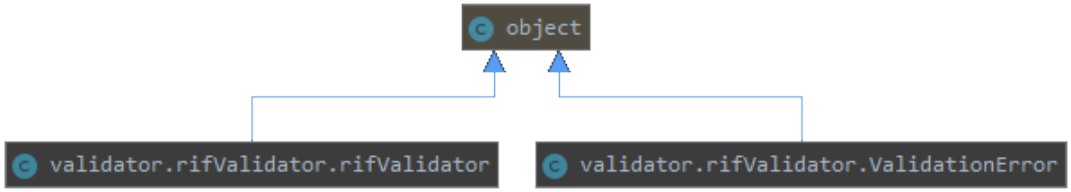

Figura 5.2: Diagrama UML de las clases del módulo `validator`.
Generado con PyCharm y yFiles.

- Se permite, para la creación de fórmulas como condiciones, el uso de fórmulas posicionales y externas, átomos de pertenencia a clase y *frames*. Es un subconjunto de los átomos definidos en el lenguaje original que no incluye ni átomos de igualdad ni de subclase.

La tercera consideración tiene como motivo aumentar la claridad y legibilidad del código escrito en el lenguaje, mientras que el resto tienen como propósito reducir la complejidad de los cómputos internos del motor.

En el análisis semántico de las reglas, se validan todas las restricciones a la estructura de los documentos RIF que son definidas para la sintaxis XML, junto con las restricciones semánticas del lenguaje y las adicionales que se han añadido.

Dentro del módulo `validator`, la clase `rifValidator` es la que se encarga de recorrer el documento RIF y comprobar que es un documento válido. Dado que el módulo trabaja con documentos en sintaxis XML, usa la librería `cElementTree` para recorrer los documentos como árboles, haciendo un recorrido en preorden por los diferentes elementos del documento.

Por otra parte, la clase `ValidationError` tiene como propósito expresar los posibles errores producidos durante la validación. Todos los métodos de `rifValidator` que forman parte del proceso de validación en sí devuelven un objeto `ValidationError`, ya que un valor `False` de su atributo booleano `err` indica que no se ha producido ningún error. Algunos ejemplos de los posibles errores que se pueden producir son encontrar una variable que no ha sido declarada anteriormente (o fuera de su ámbito de visibilidad), que un elemento XML no tenga la estructura correcta, o encontrar una constante en una posición en que su tipo de dato no se permite.

5.3. Importación de ontologías

Las bases de conocimiento que este trabajo usa son ontologías escritas en un subconjunto de OWL 2. De las características que introducen tanto RDF Schema como OWL, el motor soporta las siguientes:

- Jerarquías de clases (`rdfs:subClassOf`, `owl:Thing`) y disyunciones (`owl:disjointWith`) dos a dos.

- Equivalencia entre clases (`owl:EquivalentClass`) y equivalencia entre instancias (`owl:sameAs`, `owl:differentFrom`).
- Dominio y rango de propiedades (`rdfs:domain`, `rdfs:range`).
- Características simétrica, asimétrica, irreflexiva, disjunta e inversa (`owl:SymmetricProperty`, `owl:AsymmetricProperty`, `owl:IrreflexiveProperty`, `owl:disjointWith`, `owl:inverseOf`).

La importación de las ontologías en el trabajo se realiza con la asistencia de la librería Owlready2, una librería para la programación orientada a ontologías y su gestión [47]. El proceso consta de dos partes: el almacenamiento de la ontología y el rellenado de la *working memory*.

La primera parte del proceso ocurre en el archivo `main` y simplemente consiste en la lectura del archivo fuente de la ontología por parte de Owlready2 y su almacenamiento como objeto propio de esta librería.

El rellenado de la memoria de trabajo está programado como una función de la clase que la representa, `WorkingMemory`, situada en el módulo `datatypes`. Esta función recibe una ontología en el formato de Owlready2, y por partes, almacena las clases, las características de las propiedades, y las instancias de la ontología en diferentes diccionarios de un objeto `WorkingMemory`.

Estos objetos contienen dos diccionarios, para el almacenamiento de átomos y *frames*, y otros para almacenar las clases equivalentes y disjuntas, la jerarquía de clases, y las diferentes características que pueden tener las propiedades.

Las claves del primer diccionario son los nombres de los átomos, y tienen como valores los argumentos del átomo en cuestión. Las claves del segundo son los nombres de los elementos de la *working memory*, que tienen como valor otro diccionario cuyas claves son las propiedades de las afirmaciones RDF. Este último diccionario tiene como valores conjuntos donde se almacenan todos los predicados correspondientes a la propiedad. Así pues, dada una tripla RDF (S, P, O), `frames[S][P] = {0}`, siendo `frames` el diccionario antes mencionado. Además de *frames*, este diccionario también almacena los hechos correspondientes a membresía a una clase, pues puede ser expresada como una tripla (S, `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`, O). Las claves para el resto de diccionarios son los nombres de las clases o propiedades en sí, y el valor son las clases o características correspondientes, respectivamente.

5.4. Manipulación de reglas

Una vez un documento RIF ha sido declarado válido por el motor para trabajar con él, puede ser preparado para ello por el módulo `preparator`. Este módulo tiene dos funciones: la transformación de las reglas del documento a las estructuras de datos internas que el motor puede procesar y la optimización de su estructura, y la creación de la red Rete del documento. En esta sección se trata la primera de las dos.

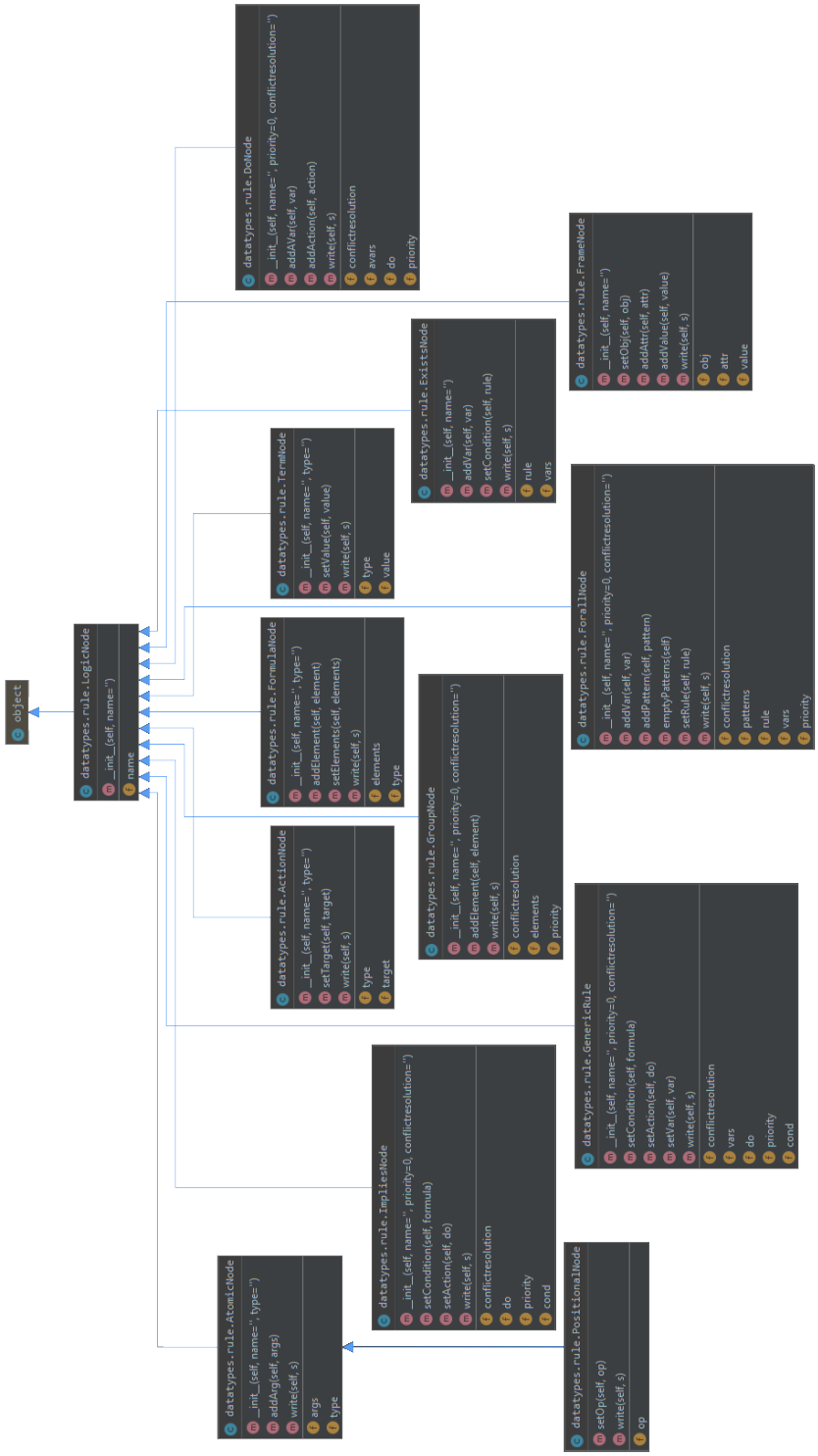

Figura 5.3: Diagrama UML de las clases referentes a reglas en `preparator`.
Generado con PyCharm y yFiles.

La manipulación de las reglas corre a cargo de la clase `ruleManipulator`, usada para leer un documento válido RIF almacenado en memoria y transformarlo en una estructura de datos interna, transformando durante el proceso las condiciones de las reglas del documento a DNF. Es importante ver que aunque el W3C delimita las reglas válidas a aquellas cuyas condiciones están escritas en forma normal disyuntiva, se ha decidido dar libertad al programador de escribir las condiciones como prefiera y hacer la transformación a DNF internamente, en vez de obligarle a trabajar siempre con esa forma.

El proceso sigue el siguiente curso:

- Crear una estructura de datos, un `GroupNode`, que contiene todo el documento RIF.
- A partir del documento, encontrar todas las reglas del documento y almacenarlas en una lista.
- Para cada regla de la lista, simplificar su estructura transformándola en un objeto `GenericRule` y pasando la fórmula lógica que es su condición a forma normal disyuntiva.

Durante la creación del `GroupNode` que representa todo el documento, se recorre el archivo RIF en profundidad, tal como se hace en la validación, y se van creando durante el mismo recorrido las estructuras adecuadas. Por ejemplo, `DoNode` representa un bloque de acciones, y `TermNode` representa términos como constantes o variables.

Para la posterior recopilación de todas las reglas en una lista, se realiza ahora un recorrido en profundidad por el `GroupNode` resultado del anterior paso. Para agilizar este recorrido, durante el paso anterior ya se asigna a cada una de las reglas atributos como su prioridad, que en el documento original vienen especificadas en los grupos.

La transformación de las condiciones de las reglas a DNF es un proceso con varios pasos que se explican en detalle por separado. El proceso recibe una regla, que puede ser un `ForallNode` (representa estructuras `Forall`), un `ImpliesNode` (representa estructuras `If Then`), o un `DoNode` (representa estructuras `Do`).

Durante la creación del `GroupNode` que representa todo el documento, se recorre el archivo RIF en profundidad, tal como se hace en la validación, y se van creando durante el mismo recorrido las estructuras adecuadas. Por ejemplo, `DoNode` representa un bloque de acciones, y `TermNode` representa términos como constantes o variables.

Para la posterior recopilación de todas las reglas en una lista, se realiza ahora un recorrido en profundidad por el `GroupNode` resultado del anterior paso. Para agilizar este recorrido, durante el paso anterior ya se asigna a cada una de las reglas atributos como su prioridad, que en el documento original vienen especificadas en los grupos.

La transformación de las condiciones de las reglas a DNF es un proceso con varios pasos que se explican en detalle por separado. El proceso recibe una regla, que puede ser un `ForallNode` (representa estructuras `Forall`), un `ImpliesNode` (representa estructuras `If Then`), o un `DoNode` (representa estructuras `Do`).

```
procedure ruleToDNF(r)
    r ← emptyForallPatterns(r, [])
    r ← simplifyStructure(r, [])
    r.cond ← moveNot(r.cond)
    r.cond ← simplifyExists(r.cond)
    r.cond ← simplifyRedundancies(r.cond)
    r.cond ← distributive(r.cond)
    return r
```

Durante la creación del `GroupNode` que representa todo el documento, se recorre el archivo RIF en profundidad, tal como se hace en la validación, y se van creando durante el mismo recorrido las estructuras adecuadas. Por ejemplo, `DoNode` representa un bloque de acciones, y `TermNode` representa términos como constantes o variables.

Para la posterior recopilación de todas las reglas en una lista, se realiza ahora un recorrido en profundidad por el `GroupNode` resultado del anterior paso. Para agilizar este recorrido, durante el paso anterior ya se asigna a cada una de las reglas atributos como su prioridad, que en el documento original vienen especificadas en los grupos.

La transformación de las condiciones de las reglas a DNF es un proceso con varios pasos que se explican en detalle por separado. El proceso recibe una regla, que puede ser un `ForallNode` (representa estructuras `Forall`), un `ImpliesNode` (representa estructuras `If Then`), o un `DoNode` (representa estructuras `Do`).

```
procedure ruleToDNF(r)
    r ← emptyForallPatterns(r, [])
    r ← simplifyStructure(r, [])
    r.cond ← moveNot(r.cond)
    r.cond ← simplifyExists(r.cond)
    r.cond ← simplifyRedundancies(r.cond)
    r.cond ← distributive(r.cond)
    return r
```

El primer paso, `emptyForallPatterns`, recorre la regla vaciando, para toda estructura `Forall` que se encuentra, sus patrones (recordar que estas estructuras tienen una serie de declaraciones de variables, una serie de condiciones llamadas patrones, y una regla). Estos son añadidos posteriormente a la regla del mismo `Forall`, repitiendo recursivamente el proceso si es necesario.

procedure EMPTYFORALLPATTERNS($r, p = \{p_1, ..., p_n\}$)
  if r is `DoNode` $\wedge\ |p| > 0$ **then**
    $r \leftarrow$ `new ImpliesNode`
    $r.cond \leftarrow$ `And(`p`)`
  if r is `ImpliesNode` $\wedge\ |p| > 0$ **then**
    $r.cond \leftarrow$ `And(`$\{p, r.cond\}$`)`
  if r is `ForallNode` **then**
    $p \leftarrow p \cup r.patterns$
    $r.patterns \leftarrow \varnothing$
    $r.rule \leftarrow$ emptyForallPatterns($r.rule$, p)
  return r

En el segundo paso, `simplifyStructure`, se transforman los tres posibles tipos de nodos que pueden representar una regla a un tipo genérico, `GenericRule`, que almacena las variables declaradas en la regla (las declaraciones de todos los `ForallNode`), y la condición y la acción de la regla original (en caso de haber). Se omiten detalles de la implementación para no dificultar la lectura del pseudocódigo.

procedure SIMPLIFYSTRUCTURE($r, v = \{v_1, ..., v_n\}$)
  if r is `DoNode` **then**
    $s \leftarrow$ `new GenericNode`
    $s.cond \leftarrow$ `And()`
    $s.act \leftarrow r.act$
    $s.vars \leftarrow v$
  if r is `ImpliesNode` **then**
    $s \leftarrow$ `new GenericNode`
    $s.cond \leftarrow r.cond$
    $s.act \leftarrow r.act$
    $s.vars \leftarrow v$
  if r is `ForallNode` **then**
    $v \leftarrow v \cup r.vars$
    $s \leftarrow$ simplifyStructure($r.rule$, v)
  return s

El tercer paso, `moveNot`, mueve las negaciones en las condiciones de las reglas lo más profundo posible, hasta que son solamente átomos los que están negados o no. Estas son las transformaciones que realiza:

$$\neg\neg p \Rightarrow p$$
$$\neg(p_1 \wedge p_2 \wedge ... \wedge p_n) \Rightarrow \neg p_1 \vee \neg p_2 \vee ... \vee \neg p_n$$
$$\neg(p_1 \vee p_2 \vee ... \vee p_n) \Rightarrow \neg p_1 \wedge \neg p_2 \wedge ... \wedge \neg p_n$$

```
procedure moveNot(cond)
    q ← cond
    while q not empty do
        e ← q.first
        if e is Not(c) then
            if c is Not(x) then
                e ← And(And(x))
            if c is And(x1, ..., xn) then
                e ← Or(Not(x1), ..., Not(xn))
            if c is Or(x1, ..., xn) then
                e ← And(Not(x1), ..., Not(xn))
        if e is And(x1, ..., xn) ∨ e is Or(x1, ..., xn) then
            q ← q ∪ {x1, ..., xn}
        if e is ∃ v1, ..., vm such that c then
            q ← q ∪ { c }
    return s
```

Por detalles de la implementación interna, la negación de una negación pasa a ser una conjunción de una conjunción en este paso. En el quinto paso esta doble conjunción se acaba eliminando para resultar en la expresión correcta ($\neg\neg p \Rightarrow p$), pero es importante notar que la doble conjunción es también equivalente así que no se pierde la equivalencia lógica.

El cuarto paso, `simplifyExists`, busca fórmulas existenciales en las condiciones de las reglas y las simplifica. Si encuentra existenciales dentro de otros existenciales, los transforma a uno solo que aúne ambos.

$$\exists x_1 \exists x_2 ... \exists x_n \, p \Rightarrow \exists x_1, ..., x_n \, p$$

```
procedure simplifyExists(cond, insideExists, v)
    if cond is FormulaNode then                    ▷ cond es And, Or o Not
        for c ∈ cond.children do
            simplifyExists(c)
            if c is ExistsNode ∧ insideExists then
                c ← c.rule
    if cond is ExistsNode then
        v' ← v ∪ cond.vars
        cond.cond ← simplifyExists(cond.cond, True, v')
    return cond
```

El quinto paso, `simplifyRedundancies`, simplifica las expresiones booleanas de las condiciones de las reglas que son redundantes, como por ejemplo conjunciones de conjunciones.

$$And(And(p_1, ..., p_n)) \rightarrow And(p_1, ..., p_n)$$
$$Or(Or(p_1, ..., p_n)) \rightarrow Or(p_1, ..., p_n)$$

Además, es en este paso que se realiza el paso a DNF de las condiciones de los cuantificadores existenciales.

---

procedure SIMPLIFYREDUNDANCIES(*cond*)
  if *cond* is `And(x1, ..., xn)` $\vee$ *cond* is `Or(x1, ..., xn)` **then**
    originalElements $\leftarrow$ `x1, ..., xn`
    for *i* $\in$ `x1, ..., xn` **do**
      if `xi` is `And(y1, ..., ym)` $\wedge$ *cond* is `And(x1, ..., xn)` **then**
        cond $\leftarrow$ `And(x1, ..., xi-1, y1, ..., ym, xi+1, ..., xn)`
      if `xi` is `Or(y1, ..., ym)` $\wedge$ *cond* is `Or(x1, ..., xn)` **then**
        cond $\leftarrow$ `Or(x1, ..., xi-1, y1, ..., ym, xi+1, ..., xn)`
    if `x1, ..., xn` has been modified **then**
      cond $\leftarrow$ simplifyRedundancies(*cond*)
  if *cond* is `FormulaNode` **then**
    for *i* $\in$ `x1, ..., xn` **do**
      `xi` $\leftarrow$ simplifyRedundancies(`xi`)
  if *cond* is `ExistsNode` **then**
    cond.cond $\leftarrow$ moveNot(*cond.cond*)
    cond.cond $\leftarrow$ simplifyRedundancies(*cond.cond*)
    cond.cond $\leftarrow$ distributive(*cond.cond*)
  return *cond*

Por último, `distributive` aplica la propiedad distributiva sobre las condiciones ahora simplificadas de las reglas, para transformarlas finalmente a DNF.

---

procedure DISTRIBUTIVE(*cond*)
  if *cond* is `Or(x1, ..., xn)` **then**
    for *i* $\in$ `x1, ..., xn` **do**
      `xi'` $\leftarrow$ distributive(`xi`).*children*
    cond $\leftarrow$ `Or(x1', ..., xn')`
  if *cond* is `And(x1, ..., xn)` **then**
    $E \leftarrow \times_{i=1}^{n}$ distributive(`xi`).*children*
    cond $\leftarrow$ `And(`E`)`
  if $\neg$ *cond* is `Or(x1, ..., xn)` $\wedge$ $\neg$ *cond* is `And(x1, ..., xn)` **then**
    cond $\leftarrow$ `Or(And(`*cond*`))`
  return *cond*

5.5. Creación de la red Rete

La segunda de las funciones del módulo `preparator` es la creación de la red para el algoritmo Rete del documento RIF, cuyas reglas en este momento ya están en forma normal disyuntiva.

La clase `ReteNetworkCreator` es la encargada de generar la red Rete a partir de una lista de reglas, para lo cual dispone de una serie de clases que representan los diferentes nodos con los que puede trabajar.

El proceso comienza creando una red vacía. La función `createReteNetwork` recorre la lista de reglas del documento generada por `ruleManipulator`, y se añaden a la red los nodos correspondientes a las comprobaciones de cada una de las conjunciones de su condición. Durante este recorrido, la función también identifica los bloques de acción (aquellas reglas sin ninguna condición) y los almacena por separado.

---

procedure CREATERETENETWORK($l = \{r_0, ..., r_n\}$)
  $network \leftarrow$ `new TopNode`
  for $r_i \in l$ **do**
    $g \leftarrow$ `new GoalNode(`$r_i.do$`,`$r_i.vars$`)`
    if $|r_i.cond| = 0$ **then**
      $network.actionblocks \leftarrow network.actionblocks \cup \{r_i\}$
    for $c \in r_i.cond.children$ **do** ▷ c es una conjunción de la condición de la regla r_i
      $network \leftarrow$ buildAnd(c, $r_i.vars$, g, $network$)
  return $network$

Antes de proseguir con la explicación de la función `buildAnd`, que añade a la red Rete parcial los nodos necesarios para representar la conjunción que recibe, es mejor explicar el rol de cada uno de los diferentes nodos que conforman una red Rete en este motor de inferencia. Estos nodos están situados en `datatypes.rete`, y sus funciones son las siguientes:

- `TopNode`: Este nodo representa el nodo raíz de la red, por el que entran los tokens, y también parte de la red Alfa. Almacena qué reglas no tienen condición (bloques de acción), qué variables de qué reglas no tienen ninguna comprobación que realizar, y la estructura de las comprobaciones a realizar. Las dos primeras están representadas como listas de nodos mientras que la tercera está representada como un conjunto de diccionarios (Figura 5.5).

  Cada uno de estos diccionarios almacena comprobaciones de un cierto tipo (átomos, *frames*, pertenencia a una clase...), y tiene como claves los elementos clave, valga la redundancia, de las comprobaciones.

  El diccionario que almacena comprobaciones de membresía a una clase almacena los nombres de las clases como clave; el que almacena comprobaciones de átomos almacena primero el nombre del átomo, luego el número de argumentos que debe tener, y por último los argumentos en sí; el que almacena *frames* almacena primero el nombre del atributo, después un número que identifica el tipo de comprobación (dada una comprobación de *frame* (S, P, O), respectivamente: la variable es O, la variable es S, o ambos son variables) y después el valor o el objeto, según corresponda y si es necesario.

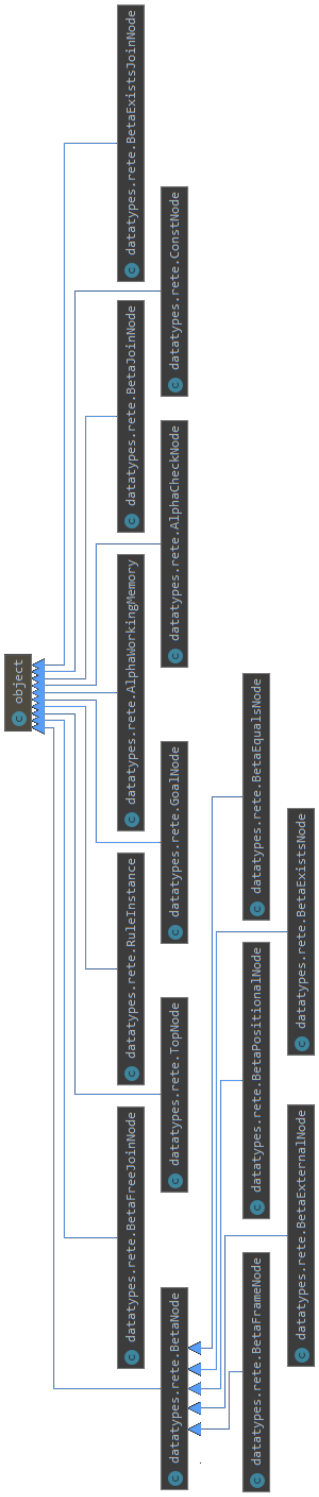

Figura 5.4: Diagrama UML de las clases referentes a la red Rete en `preparator`.
Generado con PyCharm y yFiles.

Para cada comprobación de átomo o de *frame*, sea inter-elemento o intra-elemento, se almacena dicha comprobación en la red Alfa. Es la `AlphaWorkingMemory` correspondiente, sin embargo, la que comprueba que además de tener la estructura correcta, se puede extraer del token un *mapping* variable-objeto válido.

Con esta implementación, no es necesario enviar un token concreto a todos los nodos de la red Alfa, sino que solo será enviado a realizar aquellas comprobaciones que vaya a superar.

- `AlphaWorkingMemory`: Estos nodos almacenan la lista de nodos de la red Beta a los que enviar los tokens que superan una cierta comprobación, organizados por los diferentes *mappings* variable-objeto.

  Por ejemplo, pongamos una comprobación de *frame* en que tanto objeto como valor son variables. Sin embargo, en el documento de reglas, nos encontramos dos comprobaciones diferentes de este tipo: `?x[foo ->?x]` y `?x[foo ->?y]`. Ambas comprobaciones, en el `TopNode`, estarán almacenadas como la misma comprobación e irán a parar a la misma `AlphaWorkingMemory`. No obstante, esta memoria asegurará, para superar la primera de las dos comprobaciones, que objeto y valor son iguales y por tanto se puede extraer un *mapping* válido, mientras que no lo hará en el segundo caso porque todo *mapping* será válido.

- `BetaJoinNode`: Estos nodos almacenan listas de conjuntos de objetos que han superado sus correspondientes comprobaciones y realiza comprobaciones de consistencia entre esos conjuntos de objetos, de manera que solo aquellos conjuntos de objetos que han superado todas las comprobaciones que llevan a este nodo son enviados a los siguientes nodos.

- `BetaFreeJoinNode`: `BetaJoinNode` especiales para unir conjuntos de objetos de variables que no aparecen en ninguna comprobación con el resto de conjuntos.

- `BetaExistsNode`: Colocados siempre tras todos los `BetaJoinNode`. Almacenan comprobaciones intra-elemento de aquellas variables no existencialmente cuantificadas en forma de una lista de `AlphaCheckNode`. Debido a que los elementos solo deben satisfacer estas condiciones como parte de la comprobación existencialmente cuantificada, se ha decidido no añadirlos a la red Alfa. También almacena comprobaciones inter-elemento como una lista de `BetaNode`.

- `BetaExistsJoinNode`: Nodos de funcionamiento similar a los `BetaJoinNode` que determinan qué conjuntos de elementos satisfacen al menos una de las conjunciones de la condición de la comprobación cuantificada existencialmente.

- `BetaNode`: Estos nodos, solamente usados en el interior de los `BetaExistsNode`, representan comprobaciones inter-elemento de diferentes tipos. Cada una de las clases derivadas de esta representa un tipo de comprobación inter-elemento diferente: `BetaFrameNode` para comprobaciones con *frames*, `BetaPositionalNode` para comprobaciones con átomos y `BetaExternalNode` para comprobaciones mediante llamadas a funciones.

  Los `BetaExistsNode` también son un tipo de `BetaNode`, pero por definición nunca se encontrarán en el interior de otro `BetaExistsNode`.

- `ConstNode`: Estos nodos se sitúan justo antes de los `GoalNode` y realizan todas aquellas comprobaciones de constante (aquellas en que no interviene ninguna variable) que, sin embargo, dependan del estado de la memoria de trabajo. Por ejemplo, una comprobación de *frame* para un objeto y valor concretos.

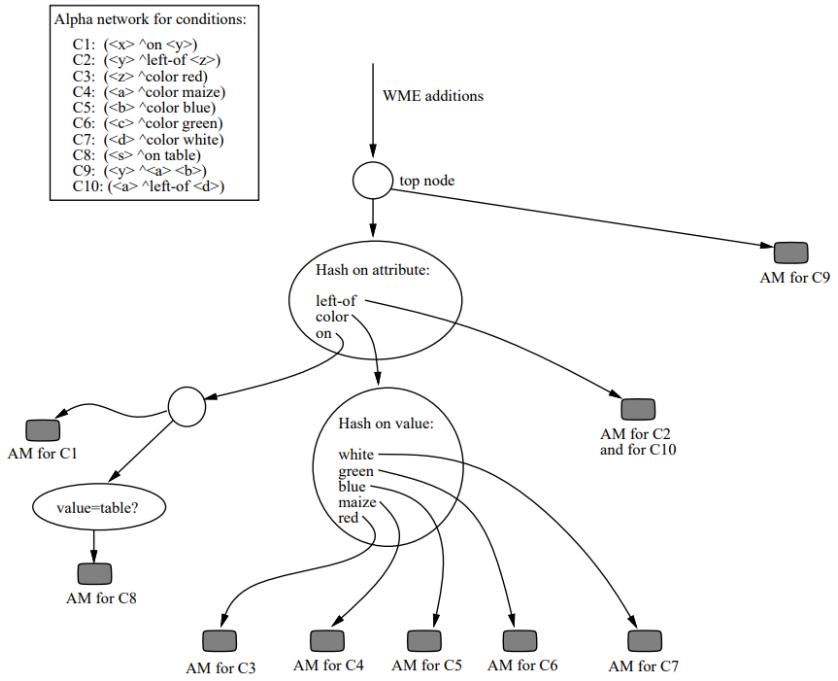

Figura 5.5: Red Alfa de ejemplo implementada con diccionarios.
Extraído de [32, p. 16].

- `GoalNode`: Estos nodos representan la satisfacción de la condición de una regla para aquellos conjuntos de elementos de la memoria de trabajo que reciben. Tienen información sobre la regla que representan (su prioridad y un identificador único), para poder enviar al motor durante la ejecución instancias (regla y elementos que satisfacen su condición), y así poder dispararlas.

La función `buildAnd` es la encargada, dada una conjunción de una regla, de crear y añadir a la red Rete todos los nodos necesarios para representarla correctamente, siguiendo los roles especificados anteriormente.

Lo primero que realiza la función es clasificar las condiciones de la conjunción en constantes, cuantificadas existencialmente y otras. Esta clasificación se realiza recorriendo cada una de las condiciones, comprobando cuál es su tipo y contando cuántas variables aparecen en esta.

Tras esto, comprueba si existe entre las comprobaciones constantes alguna que sea imposible de satisfacer, como por ejemplo `External(pred:is-literal-integer("Hola mundo!"))`. En caso de existir, cancela la creación de los nodos para la representación de la conjunción de la regla, pues será imposible que se satisfaga.

En caso de no existir ninguna comprobación imposible, continúa normalmente con el proceso añadiendo al `TopNode` las comprobaciones correspondientes, y generando los `BetaJoinNode`, `BetaExistsNode` y `ConstNode` necesarios, siendo unidos a las `AlphaWorkingMemory` u otros nodos, según corresponda.

---

procedure BUILDAND($conj = \{c_1, ..., c_n\}, vars = \{v_1, ..., v_m\}, g, network$)
  $C \leftarrow \{c_i, \text{vars}(c_i) = 0\}$ ▷ Comprobaciones constantes
  $E \leftarrow \{c_i, \text{vars}(c_i) = -1\}$ ▷ Comprobaciones cuantificadas existencialmente
  $I \leftarrow \{c_i,\ c_i \in C \wedge \text{impossible}(c_i)\}$ ▷ Comprobaciones constantes imposibles
  $O \leftarrow conj \setminus (C \cup E)$ ▷ Otras comprobaciones
  if $|I| = 0$ **then**
    $M \leftarrow \varnothing$
    for $c_i \in O$ **do**
      $m, v, neg \leftarrow \text{addAlphaCheck}(c_i, network)$ ▷ m la `AlphaWorkingMemory` de c_i
      $M \leftarrow M \cup (m, v, neg)$ ▷ v variables de c_i, neg si c_i está negada
    $V \leftarrow \varnothing$ ▷ Variables que no están en ninguna comprobación
    for $(m_i, v_i, neg_i) \in M$ **do**
      $last \leftarrow \text{addBetaJoin}(m_i,\ v_i,\ neg_i,\ last)$
      $V \leftarrow V \cup v_i$
    for $v_i \notin V$ **do**
      $last \leftarrow \text{addFreeBetaJoin}(v_i,\ last)$
    for $c_i \in E$ **do**
      $last \leftarrow \text{addExistentialCheck}(c_i, network, vars,\ last)$
    $c \leftarrow$ `new ConstNode(`C`)`
    $last$.setChild(c)
    c.addChild(g)
  return $network$

Notar que en esta función, dada una comprobación c, vars(c) corresponde al número de variables diferentes que intervienen en c (excepto si es una comprobación existencialmente cuantificada, en cuyo caso vars(c)$= -1$) e impossible(c) es cierto solo cuando c es una comprobación imposible de satisfacer.

También que aunque en el pseudocódigo anterior se omite, se gestionan aspectos como cuál es el último nodo creado de la red Beta (y si este existe siquiera), o que el primer `BetaJoinNode`, al no estar conectado con ningún otro, recibe todas sus listas de `AlphaWorkingMemory` (a diferencia del resto, que recibe una lista de `AlphaWorkingMemory` y otra del `BetaJoinNode` con el que está conectado).

En aras de la brevedad, se omiten los pseudocódigos para las funciones `addAlphaCheck` y `addBetaJoin` y `addFreeBetaJoin`.

En pocas palabras, `addAlphaCheck` construye una tupla con toda la información necesaria para llevar a cabo la comprobación, y la envía al `TopNode` para que este añada al diccionario correcto (según el tipo de comprobación) la clave referente a la comprobación (o claves en el caso de *frames* o átomos). Después, se da como valor de esta clave una `AlphaWorkingMemory` (si la comprobación no existía todavía). La `AlphaWorkingMemory`, entonces, añade la lista de variables de la comprobación.

La función `addBetaJoin` (que realmente es un proceso dentro de `buildAnd`), por su parte, crea un `BetaJoinNode` y lo enlaza con la `AlphaWorkingMemory` correspondiente y el `BetaJoinNode` anterior, excepto si es el primer `BetaJoinNode`, en cuyo caso lo enlaza con las `AlphaWorkingMemory` de las dos primeras comprobaciones.

`addExistentialCheck`, dada una comprobación existencialmente cuantificada, recorre cada una de las conjunciones de su condición, en DNF. Para cada una de las conjunciones, comprueba que no existen comprobaciones imposibles de satisfacer y añade al `TopNode` las comprobaciones intra-elemento de las variables cuantificadas existencialmente. El resto de comprobaciones son almacenadas en el `BetaExistsNode` que se crea para la conjunción, para ser comprobadas a la vez cuando un conjunto de elementos llega al nodo. Además, se crea un `BetaExistsJoinNode`, que enlaza a todos los nodos de las diferentes conjunciones y gestiona el envío de los conjuntos de elementos que superan la comprobación a los nodos que corresponda.

---

procedure ADDEXISTENTIALCHECK($c, network, vars = \{v_1, ..., v_m\}$)
  $joinnode \leftarrow$ `new BetaExistsJoinNode`
  for $k_i = \{c_1, ..., c_n\} \in c.cond$ **do** ▷ $c.cond$ es una DNF y k_i es una conjunción
    $A \leftarrow \{c_i, \text{vars}(c_i) = 1 \wedge \hat{v} \in c.vars\}$ ▷ Sea $\hat{v}$ la variable de la comprobación
    $B \leftarrow \{c_i, \text{vars}(c_i) \geqslant 1 \wedge c_i \notin A\}$
    $C \leftarrow \{c_i, \text{vars}(c_i) = 0\}$ ▷ Comprobaciones constantes
    $I \leftarrow \{c_i,\ c_i \in C \wedge \text{impossible}(c_i)\}$ ▷ Comprobaciones constantes imposibles
    if $|I| = 0$ **then**
      $M \leftarrow \varnothing$
      for $c_i \in A$ **do**
        $m, v, neg \leftarrow \text{addAlphaCheck}(c_i, network)$
        $M \leftarrow M \cup (m, v, neg)$
      for $j_i \in J$ **do**
        if j_i has not been used **then**
          $network.freejoins \leftarrow network.freejoins \cup \{j_i\}$
      $node \leftarrow$ `new BetaExistsNode(`$B \cup C$`)` ▷ $B \cup C$ se almacena dentro del mismo nodo
      $V \leftarrow \varnothing$ ▷ Variables que no están en ninguna comprobación
      for $(m_i, v_i, neg_i) \in M$ **do**
        mem.addChild(v_i, $node$, neg_i)
        $V \leftarrow V \cup v_i$
      for $v_i \notin V$ **do**
        $last \leftarrow \text{addFreeWM}(v_i, node)$
      $node.children \leftarrow \{joinnode\}$
  return $network$

Notar que `addFreeWM` añade una `AlphaWorkingMemory` directamente conectado con el `TopNode` y la enlaza con el `BetaExistsNode` en cuestión.

5.6. Ejecución

Dentro del sistema, la ejecución es el proceso en que dados una ontología y un documento de reglas, el motor los usa para inferir nuevo conocimiento y extraer conclusiones. Este proceso es llevado a cabo por el módulo `execution` donde, en vez de haber definida una clase que lleva a cabo el proceso, se recopilan una serie de funciones en el archivo `engine.py` que permiten realizar la ejecución.

El proceso comienza con la creación y rellenado de un objeto `WorkingMemory`, una clase situada en `datatypes.workingmemory` cuya función es almacenar los hechos de la memoria de trabajo de manera ordenada y fácilmente accesible. Este proceso se explica en la sección Importación de ontologías.

Tras esto, que se realiza todavía en el menú principal, se llama a la función `execute`, que recibe un objeto `WorkingMemory`, una red Rete (un `TopNode`), y la lista de nodos objetivo (`GoalNode`) por separado.

Antes de continuar con el proceso, es necesario comentar brevemente de dos clases, `ConflictSet` y `RuleInstance`.

Los objetos `RuleInstance` están formados por el identificador de una regla, su prioridad, y un conjunto de elementos de la *working memory* que satisfacen su condición. Estos almacenan toda la información necesaria para ser ordenados correctamente por el conjunto de conflicto, y son generados por los `GoalNode` cuando reciben un conjunto de elementos.

La clase `ConflictSet` representa un *conflict set*, y está implementado como una cola de prioridad. En la cola de prioridad, las instancias se ordenan en primer lugar según su prioridad, y en segundo lugar según cuándo se han ejecutado por última vez. En caso de empate, también se ordena según el identificador de la regla. La clase tiene, además, funciones para añadir, eliminar y modificar elementos de dicha cola. La gestión sobre el tiempo pasado desde la última ejecución de cada regla se realiza en la misma función `execute`.

La función `execute` comienza rellenando el *conflict set* con los bloques de acción del documento. Seguidamente se envían todos los hechos de la memoria de trabajo recién rellenada a la red Rete para determinar qué reglas pueden ejecutarse con qué elementos, generar las `RuleInstance` correspondientes y actualizar el conjunto de conflicto.

Tras esta inicialización se entra en un bucle en que se ejecuta la primera regla del conjunto de conflicto, eliminándola así del conjunto, se marca la `RuleInstance` correspondiente como ya disparada (recordemos que cada conjunto regla-elementos único solo puede ser disparado una vez en una ejecución). Los cambios en la memoria de trabajo provocados por la regla son enviados a la red Rete, que a su vez devuelve los adiciones y eliminaciones de `RuleInstance` a realizar en el conjunto de conflicto. Tras modificar también acordemente en el *conflict set* los tiempos pasados

desde la última ejecución de la regla que se acaba de ejecutar, comienza una nueva iteración del bucle. Este dura hasta que el conjunto de conflicto queda vacío.

---

procedure EXECUTE($wm = \{f_1, ..., f_n\}, network, g = \{g_1, ..., g_m\}$)
  $cs \leftarrow$ `new ConflictSet`
  cs.add($network.actionblocks$)
  for $f_i \in wm$ **do**
    $network$.sendFact(f_i) ▷ Se envía f_i a la red Rete
  while $|cs| > 0$ **do**
    $ri \leftarrow cs.first$
    $goal \leftarrow g_i$, $g_i.id = ri.id$ ▷ Se escoge el `GoalNode` correspondiente a la `RuleInstance`
    $changes \leftarrow goal$.fire($ri.wmelements, wm$) ▷ Disparar r_i devuelve los cambios en wm
    for $t_i \in changes$ **do**
      $network$.sendFact(f_i)

En el pseudocódigo se omite la gestión de la información de las reglas para que quede más claro, pero esta se realiza en la función mediante un diccionario llamado `ruleinfo`, que tiene por claves los identificadores y por valores sus prioridades, la iteración del bucle en que se ejecutaron por última vez y los conjuntos de elementos para los que todavía pueden ser disparadas. Cuando una regla se dispara, el conjunto de elementos con que lo ha hecho se elimina del vector, y se actualiza la iteración de última ejecución. Con la información de este vector se pueden modificar los elementos del *conflict set*, que debido a su implementación necesita que se le pase una copia exacta del elemento a modificar.

El envío de tokens y conjuntos de objetos por la red Rete se realiza mediante la función `sendFact`. Cada tipo de nodo que interviene en el proceso tiene su propia función `sendFact`, cuyo funcionamiento varía ligeramente entre nodos.

Los tokens se representan mediante los objetos `TokenFact`, que almacenan el tipo de hecho que representan, si se trata de la aserción o la eliminación de un hecho, y el hecho en sí.

En el `TopNode`, primero se comprueba el tipo del token recibido, y tras ello se comprueba en el diccionario correspondiente si existe alguna comprobación de misma estructura.

---

procedure SENDFACT($token$)
  if $token.type = atom$ **then**
    checkAtom($token$)
  if $token.type = frame$ **then**
    if $token.fact[1] =$ `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` **then**
      checkMembership($token$)
    if $token.fact[1] \neq$ `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` **then**
      checkFrame($token$)

```
procedure checkAtom(token)
    op ← token.fact[0]
    args ← token.fact[1]
    for c ∈ positional[op][|args|] do
        pass ← True
        for e ∈ c do                          ▷ Comprobar que token sigue la estructura de c
            if ¬(e is None) then
                passes ← e = args[i]
            if ¬passes then
                break
        positional[op][|args|][c].sendFact(token, args, token.add)
```

```
procedure checkMembership(token)
    subj ← token.fact[0]
    obj ← token.fact[2]
    if ∃ membership[obj] then
        membership[obj].sendFact(token, (None, obj), token.add)
```

`checkFrame` funciona análogamente a `checkMembership` con la diferencia de que se comprueban las claves `1`, `2` y `3` cuando el atributo del token se encuentra en el diccionario (más detalle en la explicación del `TopNode` del apartado Creación de la red Rete). Así pues, dado un token que representa la afirmación RDF (S, P, O), se comprueban `frame[`P`][1][`S`]`, `frame[`P`][2][`O`]` y `frame[`P`][3]`, para los tres tipos posibles de comprobaciones.

La función `sendFact` en las `AlphaWorkingMemory` tiene dos funciones. En primer lugar comprueba, para cada una de las disposiciones de variables que almacena, si el token que ha recibido es compatible con esta. En segundo lugar, envía el token a los nodos de la red Beta correspondientes a aquellas disposiciones compatibles con el token. Referirse a la explicación de las `AlphaWorkingMemory` de la sección Creación de la red Rete para un ejemplo de diferentes disposiciones de variables.

```
procedure sendFact(token, template, add)
    if token.type = frame then
        for d_i ∈ ids do                          ▷ Cada d_i es una disposición diferente
            passes ← True
            objs ← {}
            k = 0
            if template[0] is None then
                objs ← {d_i[k] ↦ token.fact[0]
                k = 1
            if template[1] is None then
                if d_i[k] ∈ objs ∧ token.fact[2] ≠ token.fact[0] then
                    passes ← False
                if d_i[k] ∉ objs then
                    objs ← objs ∪ {d_i[k] ↦ token.fact[2]}
            if passes then
                for node_j, index_j ∈ ids[d_i] do
                    node_j.sendFact(objs, index_j, add)
    if token.type = atom then
        for d_i ∈ ids do
            for e_j ∈ template do
                passes ← True
                objs ← {}
                k = 0
                if e_j is None then
                    if d_i[k] ∉ objs then
                        objs ← objs ∪ {d_i[k] ↦ token.fact[1][j]}
                    if d_i[k] ∈ objs then
                        passes ← objs[d_i[k]] = token.fact[1][j]
                    k = k + 1
            if passes then
                for node_j, index_j ∈ ids[d_i] do
                    node_j.sendFact(objs, index_j, add)
```

La función `sendFact` para los `BetaJoinNode` (que por definición tienen dos listas de conjuntos), cuando recibe un *mapping* objeto-variable, lo añade o elimina de la lista correspondiente.

Cuando un conjunto de elementos está en una lista, es porque satisface la comprobación o comprobaciones correspondientes a esa lista (o no satisface la comprobación, si esta está negada). Así pues, el nodo deja pasar aquellos conjuntos de elementos que pasan las comprobaciones correspondientes a la otra lista, y que son compatibles con el *mapping* que ha recibido. Si la lista a la que se añade el *mapping* corresponde a una comprobación negada, se realizará justo lo contrario a lo dicho.

```
procedure sendFact(token = {n_1 ↦ e_1, ..., n_n ↦ e_n}, index, add)
    if add ∧ token ∉ tokenlists[index] then
        tokenlists[index] ← tokenlists[index] ∪ token
        otherindex ← 1 − index
        if ¬neglists[index] ∧ ¬neglists[otherindex] then
            C ← {t_i, t_i ∈ tokenlists[otherindex] ∧ token ⩽ t_i}
            for t_i ∈ C do
                child.sendFact(token ∪ t_i, child.index, True)
        if ¬neglists[index] ∧ neglists[otherindex] then
            C ← {t_i, t_i ∈ tokenlists[otherindex] ∧ token ⩽ t_i}
            if |C| = 0 then
                child.sendFact(token, child.index, True)
        if neglists[index] ∧ neglists[otherindex] then
            C ← {t_i, t_i ∈ tokenlists[otherindex] ∧ token ⩽ t_i}
            if |C| = 0 then
                child.sendFact(token ∪ t_i, child.index, False)
    if ¬add ∧ token ∈ tokenlists[index] then
        tokenlists[index] ← tokenlists[index] ∪ token
        otherindex ← 1 − index
        if ¬neglists[index] ∧ ¬neglists[otherindex] then
            C ← {t_i, t_i ∈ tokenlists[otherindex] ∧ token ⩽ t_i}
            for t_i ∈ C do
                child.sendFact(token ∪ t_i, child.index, False)
        if ¬neglists[index] ∧ neglists[otherindex] then
            C ← {t_i, t_i ∈ tokenlists[otherindex] ∧ token ⩽ t_i}
            if |C| = 0 then
                child.sendFact(token, child.index, False)
        if neglists[index] ∧ neglists[otherindex] then
            C ← {t_i, t_i ∈ tokenlists[otherindex] ∧ token ⩽ t_i}
            if |C| = 0 then
                child.sendFact(token ∪ t_i, child.index, True)
```

Notar que se define la función $a \leqslant b$ en conjuntos de objetos como que el subconjunto de claves de a que se puede encontrar en b es compatible. Es decir, que los valores para esas claves son iguales. Notar también que `neglists` contiene qué listas se corresponden a comprobaciones negadas, y `tokenlists` es la lista de listas de *mappings*.

Los `BetaFreeJoinNode` reciben todo objeto que llega como parte de un token al `TopNode`. Su `sendFact` funciona de manera similar a los `BetaJoinNode`, pero dado que por definción los *mappings* de listas diferentes nunca compartirán variable (es decir, dos listas diferentes no contendrán valores para la misma variable) simplemente se envía al siguiente nodo la combinación del conjunto de objetos que recibe con todos los de la otra lista.

Los `BetaExistsNode`, cada vez que reciben un conjunto de elementos para ciertas variables, lo unen con el producto cartesiano de los elementos candidatos a tomar valor del resto de variables y, para cada uno de los conjuntos resultantes, realizan todas las comprobaciones pertinentes a la conjunción correspondiente al nodo de la comprobación existencialmente cuantificada a la que pertenecen.

---

procedure SENDFACT($objs = \{n_1 \mapsto e_1, ..., n_n \mapsto e_n\}, add, index$)
  if $add \wedge objs \notin$ `tokenlists` **then**
    `tokenlists[`$index$`]`$\leftarrow$ `tokenlists[`$index$`]`$\cup\{subj\}$
  if $\neg add \wedge objs \in$ `tokenlists` **then**
    `tokenlists[`$index$`]`$\leftarrow$ `tokenlists[`$index$`]`$\setminus\{subj\}$
  for $p \in \bigtimes_{i=1}^{|tokenlists \setminus tokenlists[index]|} L$ **do**
    $p \leftarrow p \cup objs$
    $passed \leftarrow p \in$ `results`
    $passes \leftarrow \bigwedge_{j=1}^{|existsvars|}$ `existsvars[`j`]` $\in p \wedge \bigwedge_{j=1}^{|checks|}$ check(`checks[`j`]`)
    if $(\neg$`not`$\wedge\neg passed \wedge passes) \vee ($`not`$\wedge\neg passed \wedge \neg passes)$ **then**
      for $i_i, node_i \in$ `children` **do**
        $node_i$.sendFact(p,`True`,i_i)
    if $(\neg$`not`$\wedge passed \wedge \neg passes) \vee ($`not`$\wedge passed \wedge passes)$ **then**
      for $i_i, node_i \in$ `children` **do**
        $node_i$.sendFact(p,`False`,i_i)

Notar que en la asignación a *passes*, la primera de las dos operaciones de la conjunción se refiere a que p debe tener una asignación para cada una de las variables cuantificadas existencialmente.

Por su parte, cada una de las comprobaciones internas del `BetaExistsNode` se realizan en el correspondiente `BetaNode` y, aunque ligeramente diferentes, consisten en uno o más accesos a la memoria de trabajo para comprobar la existencia o no existencia de un hecho (excepto los `BetaExternalNode`, que consisten en la evaluación de una función).

Los `BetaExistsJoinNode` son los que reciben aquellos conjuntos de elementos que envían los `BetaExistsNode`, y que consecuentemente envían a sus hijos en la red Beta.

Por último, el `GoalNode` añade o elimina de sus memorias los conjuntos de elementos que recibe, y realiza el producto cartesiano de la misma manera que los `BetaExistsNode` para generar los `RuleInstance` correspondientes.

```
procedure sendFact(objs = {n_1 ↦ e_1, ..., n_n ↦ e_n}, add, index)
    if add then        ▷ tokenlists son las diferentes listas de elementos que ha recibido el nodo
        tokenlists[index] ← tokenlists[index] ∪ {subj}
    if ¬add then
        tokenlists[index] ← tokenlists[index] \ {subj}
    L ← tokenlists \ tokenlists[index]
    for p ∈ ×_{i=1}^{|L|} L do
        p ← p ∪ objs
        if |p| = |vars| then
            if add ∧ p ∉ results then
                results ← results ∪ p
                addToConflictSet(new RuleInstance(priority,id,p))
            if ¬add ∧ p ∈ results then
                results ← results \ p
                removeFromConflictSet(new RuleInstance(priority,id,p))
```

El disparo de una regla se realiza a través de la función `fire` de los `GoalNode`, que llaman a la función `fire` de `execution.engine`.

`fire`, en primer lugar, crea todas las variables de acción. A las que tiene que dar valor se lo da, accediendo a la memoria de trabajo. A aquellas que representan nuevos objetos, les da como valor un `BNode`, un objeto de la librería RDFLib que representa un recurso anónimo (*blank node*) [37] al que en este motor posteriormente se le pueden añadir propiedades como a cualquier otro objeto.

Tras ello, lleva a cabo las acciones que se detallan en la regla (aserciones, eliminaciones, modificaciones y ejecuciones). Estas acciones (exceptuando las ejecuciones) se llevan a cabo en la *working memory*, que tras realizarlas, devuelve la lista de cambios en la memoria de trabajo en forma de tokens que hay que enviar a la red Rete.

La función `eval`, por su parte, recibe un nodo representando una función que evaluar, y un mapeo con todos nos nombres de variable y sus valores que puede contener la función. Tras sustituir las variables en los argumentos por sus valores correspondientes, y evaluar recursivamente las funciones que también figuraran como argumento, evalúa la función. Las funciones están almacenadas en un diccionario cuyas claves son sus nombres de función, la mayoría de ellas como funciones lambda, y son llamadas por `eval` cuando es necesario.

procedure FIRE($actions = \{a_1, ..., a_n\}, actionvars = \{v_1, ..., v_m\}, mapping, wm$)
  $avmapping \leftarrow \varnothing$
  for $v_i \in actionvars$ **do**
    if v_i is `New()` **then**
      $avmapping \leftarrow avmapping \cup \{i \mapsto$ `new BNode`$\}$
    if v_i is not `New()` **then**
      $avmapping \leftarrow avmapping \cup \{i \mapsto wm.frames[mapping[v_i.object]][v_i.attribute]\}$
  $T \leftarrow \varnothing$
  for $a_i \in actions$ **do**
    for $m \in \times_{i=1}^{|avarmapping|} avarmapping$ **do**
      for $av_i \in m$ **do**
        $mapping(v_i.value) \leftarrow av_i$
    if $a_i.type = assert$ **then**
      $t \leftarrow wm.\text{assert}(a_i.target)$
    if $a_i.type = retract$ **then**
      $t \leftarrow wm.\text{retract}(a_i.target)$
    if $a_i.type = modify$ **then**
      $t \leftarrow wm.\text{modify}(a_i.target)$
    if $a_i.type = execute$ **then**
      $\text{eval}(a_i.target, mapping)$
    $T \leftarrow T \cup t$
  return T

Capítulo 6

Estudio del rendimiento

Con el objetivo de evaluar el rendimiento del motor que se ha desarrollado en relación a otros sistemas de similar utilidad, se ha llevado a cabo un pequeño estudio.

6.1. Metodología

El estudio realizado compara los tiempos de ejecución del motor con los de los programas CLIPS y JessRules en ejecuciones de documentos de reglas similares (adaptados al formato de cada motor) con las mismas ontologías (también adaptadas según el formato de cada motor).

CLIPS es un lenguaje de programación basado en reglas que se usa para la creación de sistemas basados en el conocimiento implementado en C [9]. Para medir su rendimiento se ha usado el entorno de desarrollo que prové, ya que tiene integradas herramientas para la medición de tiempo.

Por su parte, JessRules es una librería para Java que pone a disposición del usuario un motor de inferencia que usa la *Java Virtual Machine* (JVM) [48]. Se ha medido el tiempo de ejecución de JessRules usando funciones por defecto de Java.

Se han medido tiempos de ejecución para cinco documentos de reglas diferentes, cada uno centrado en características diferentes de los motores, con cuatro ontologías de diferentes tamaños (50, 250, 500 y 1000 instancias). Las ontologías contienen personas con un tipo (hombre o mujer), un nombre de entre tres posibles por género y algunas de ellas, un marido o esposa. Los documentos de reglas previamente mencionados contienen condiciones que aluden a estas propiedades.

La construcción de las reglas de los documentos se ha basado en los *Berlin SPARQL Benchmarks* (BSBM), una serie de pruebas estándar para el estudio del rendimiento de consultas en sistemas de almacenamiento que implementan el protocolo SPARQL [49]. Estos *benchmarks* introducen casos de uso tanto de exploración de bases de conocimiento como actualizaciones de las mismas [49][50].

A continuación, una pequeña descripción de cada uno de los documentos:

- **Test 1**: Escribir por pantalla todas aquellas instancias que tienen marido o esposa.

- **Test 2**: Realizar una aserción de átomo (o equivalente) por cada persona que tiene marido o esposa.
- **Test 3**: Realizar una aserción de átomo (o equivalente) por cada persona que tiene marido o esposa y un nombre concreto.
- **Test 4**: Realizar una aserción de átomo (o equivalente) por cada persona que tiene marido o esposa y un nombre concreto, donde la aserción se realiza sobre una variable de acción (o equivalente).
- **Test 5**: Eliminar de la memoria de trabajo aquellas instancias que tienen un nombre concreto, y posteriormente realizar una aserción por cada hombre con un nombre concreto que tiene marido o esposa.

Para obtener cada uno de los tiempos, se han realizado tres ejecuciones y se ha calculado la media.

Adicionalmente, también se calculan los tiempos de ejecución corrigiendo la influencia del lenguaje de programación en que los sistemas están escritos, para poder obtener una comparación más justa. Para ello, se ha usado el trabajo de Pereira et al. [51], en el que se estudia el rendimiento de 27 lenguajes comúnmente usados en cuanto a uso energético, tiempo de ejecución y uso de memoria. Para ello, usan los *benchmarks* proporcionados por *The Computer Language Benchmarks Game* [52] para la comparación de lenguajes. El resultado (y los valores que se usarán en el cálculo) es que Java es de media 1.89 veces más lento que C, mientras que Python lo es 71.9 veces.

6.2. Resultados

De los resultados obtenidos (Tabla 6.2) se pueden extraer varias conclusiones. Notar que el llamado pyRIF en las tablas es el motor de inferencia desarrollado en este trabajo.

La primera de ellas es que el tiempo de ejecución en pyRIF parece escalar según la cantidad de condiciones en la red Rete. En ontologías de 1000 instancias, los *tests* con condiciones más complejas tienen hasta el doble de tiempo de ejecución, mientras que el resto de motores siguen obteniendo tiempos de ejecución similares. También escala mucho en función del número de instancias de la ontología, pero esto puede explicarse por el rendimiento de Python en comparación con el de los otros lenguajes.

Se puede observar que la escritura por pantalla no es un factor relevante en el motor (aunque se puede observar), pues la diferencia entre los tiempos de las dos primeras pruebas es bastante pequeña, y estas solo difieren en si hay escritura por pantalla o una aserción por instancia que cumple la condición de la regla.

	Test 1				Test 2			
	50	250	500	1000	50	250	500	1000
pyRIF	0.006	0.026	0.086	0.273	0.002	0.021	0.065	0.223
CLIPS	0.037	0.211	0.379	0.760	0.009	0.014	0.020	0.037
JessRules	0.016	0.024	0.023	0.049	0.013	0.027	0.025	0.047

	Test 3				Test 4			
	50	250	500	1000	50	250	500	1000
pyRIF	0.003	0.025	0.102	0.446	0.003	0.025	0.103	0.457
CLIPS	0.009	0.014	0.032	0.034	0.010	0.016	0.025	0.035
JessRules	0.019	0.036	0.025	0.042	0.034	0.042	0.041	0.041

	Test 5			
	50	250	500	1000
pyRIF	0.005	0.034	0.130	0.498
CLIPS	0.010	0.014	0.021	0.036
JessRules	0.031	0.027	0.024	0.035

Tabla 6.1: Tiempos de ejecución (en segundos) por *test*, motor de inferencia y tamaño de ontología. Elaboración propia.

Otra característica cuyo peso en el rendimiento parece negligible es la del uso de variables de acción en las producciones. La prueba 4, la que las usa, tiene los mismos tiempos que la prueba 3 (solo difieren en el acceso a atributos de un elemento que satisface las condiciones de la regla). En cambio, parece que la mayor cantidad de accesos a la memoria de trabajo efectuados por el *test* 5 se reflejan en unos tiempos de ejecución algo superiores a los del resto.

Hay que notar, de todas formas, que las diferencias temporales se ven enfatizadas por el peor rendimiento de Python.

En cuanto a la comparación con CLIPS y JessRules, hay dos aspectos a destacar. En primer lugar, que para los tamaños de ontología pequeños (50 y 250 instancias), el motor tiene menores tiempos de ejecución que el resto de manera consistente. En segundo lugar, como se decía previamente, el motor presenta un escalado mucho más exagerado según el número de condiciones en la red Rete que el resto de motores, donde el tiempo de ejecución no parece verse afectado por variaciones tan pequeñas en el número de condiciones.

Una curiosidad a destacar es el mal rendimiento de CLIPS en la prueba 1, la única que tiene escrituras por pantalla. La explicación es sencilla: el entorno de desarrollo que usa es muy ineficiente en cuanto al *output*. Otro detalle a notar es que las pequeñas inconsistencias en los tiempos de JessRules son causadas por el uso de la JVM, que provoca una mayor variabilidad en los tiempos de ejecución que los del resto de motores.

	Test 1				Test 2			
	50	250	500	1000	50	250	500	1000
pyRIF (Python)	0.0001	0.0004	0.0012	0.0038	0.0001	0.0003	0.0009	0.0031
CLIPS (C)	0.037	0.211	0.379	0.760	0.009	0.014	0.020	0.037
JessRules (Java)	0.0083	0.0125	0.0120	0.0261	0.0067	0.0145	0.0134	0.0249

	Test 3				Test 4			
	50	250	500	1000	50	250	500	1000
pyRIF (Python)	0.0001	0.0003	0.0014	0.0062	0.0001	0.0005	0.0018	0.0064
CLIPS (C)	0.009	0.014	0.032	0.034	0.010	0.016	0.025	0.035
JessRules (Java)	0.0102	0.0140	0.0323	0.0343	0.01	0.0160	0.0247	0.035

	Test 5			
	50	250	500	1000
pyRIF (Python)	0.0001	0.0005	0.0018	0.0069
CLIPS (C)	0.010	0.014	0.021	0.036
JessRules (Java)	0.0166	0.0141	0.0125	0.0183

Tabla 6.2: Tiempos de ejecución (en segundos) corrigiendo la influencia del rendimiento del lenguaje. Elaboración propia.

En definitiva, lo más importante que se puede extraer del estudio es que en el motor, la mayor influencia sobre el tiempo de ejecución (además del número de elementos con el que se trabaja) parece tenerla el tamaño de la red Rete. Además, parece que este tamaño tiene una influencia mayor que la que debería tener, provocando un mayor escalado que en el resto de motores.

Por otra parte, cuando se aplica la corrección de la influencia del lenguaje (recordemos, Java es más lento que C de media 1.89 veces, y Python 71.9 veces), los tiempos de ejecución del motor son mucho menores que los de CLIPS y JessRules. Es decir, si los tres motores estuvieran programados en un lenguaje común (en este caso C), la implementación de pyRIF sería la más rápida. Sin embargo, se puede observar que incluso en este caso el motor escala peor que CLIPS y JessRules: los tiempos en la ontología de 1000 instancias son unas 3 o 4 veces aproximadamente mayores que los obtenidos en la ontología de 50 instancias en los casos de los dos últimos motores, mientras que en pyRIF puede llegar hasta las 60 veces o más. Así pues, con tamaños de ontologías suficientemente grandes, pyRIF acabaría siendo más lento. También se puede ver que el problema del escalado con respecto al tamaño de la red Rete sigue ahí.

De todas formas, hay que recordar que los valores usados en la corrección son experimentales, resultado de un solo estudio. Y aunque los valores reales estén probablemente cerca de los que resultaron del estudio, hay que tomarlos como puramente orientativos. Así pues, los tiempos corregidos han de tratarse como una aproximación.

Capítulo 7

Conclusiones

En este trabajo de fin de grado se ha presentado conocimiento sobre lógica de primer orden, web semántica y sus tecnologías, sistemas basados en el conocimiento e inferencia lógica. Muchos de estos temas son necesarios para el desarrollo de *software* para la web semántica, y el conocimiento de todos ellos es imprescindible cuando se habla de la implementación de motores de inferencia.

Este conocimiento ha sido aplicado en este trabajo mediante el desarrollo e implementación de un motor de inferencia en Python basado en el dialecto de reglas de producción del lenguaje RIF, y OWL, lenguaje para la descripción de ontologías.

En primer lugar, el motor consta de un traductor que permite el paso de documentos de reglas escritos en la sintaxis de presentación de RIF, mucho más cómoda para la creación de reglas, a documentos equivalentes en la sintaxis XML del mismo lenguaje, que es la sintaxis normativa del lenguaje y la que el motor usa para trabajar con los documentos de reglas en el resto de secciones. También posee un módulo validador de documentos de reglas, que permite comprobar si un documento en la sintaxis XML antes mencionada se atañe a la redefinición del dialecto de reglas de producción de RIF que se ha efectuado para ser compatible con las necesidades de este motor.

En cuanto al motor de inferencia propiamente dicho, se ha programado una implementación del algoritmo Rete, de manera que a partir de un documento de reglas válido, el programa genera una red Rete capaz de realizar *mathing* para encontrar qué objetos de la base de conocimiento cumplen las reglas en cuestión. En adición, se ha implementado la importación de ontologías en lenguaje OWL, con compatibilidad con algunas de las características de este. La importación permite usar bases de conocimiento escritas en ese lenguaje para realizar inferencia junto a un documento de reglas.

Por último, por supuesto, se ha programado la ejecución del motor de inferencia propiamente dicha: el proceso en que, dado un documento de reglas en forma de red Rete y una ontología, el motor genera nuevo conocimiento. Se ha implementado el funcionamiento de la red Rete, la memoria de trabajo, un *conflict set* y una estrategia de resolución de conflictos.

Para el cómodo uso de los módulos que forman el sistema, se ha programado una sencilla interfaz de comandos. Aunque se marcaba como objetivo inicialmente el desarrollo de una interfaz gráfica, esto no ha sido posible. De todas formas, no es una parte fundamental ni necesaria del trabajo e incluso se definía como objetivo opcional.

Aun así, se puede decir que se ha cumplido el objetivo principal propuesto, el desarrollo del motor de inferencia. Aunque se han tenido que redefinir o limitar ciertas secciones del lenguaje RIF para adaptarlo a las necesidades del proyecto, el resultado final es un sistema que realiza inferencia lógica a partir de reglas y ontologías.

Se puede decir lo mismo de los objetivos relacionados con el aprendizaje y la teoría tras el trabajo, pues se ha podido encontrar y asimilar todo el conocimiento necesario para su desarrollo.

Por último, en cuanto a la corrección del motor, no ha sido posible garantizar totalmente el correcto funcionamiento debido a la gran extensión del proyecto. Por poner un ejemplo, el motor tiene definidas más de 60 funciones preparadas para ser usadas en los documentos de reglas como funciones definidas externamente. Solo el *testing* exhaustivo de esa pequeña porción de una de las funciones del motor llevaría una cantidad importante de tiempo. Así pues, aunque se puede afirmar que el motor funciona en los casos generales, pueden existir casos límite para los que el programa de resultados incorrectos. Se pueden encontrar las pruebas que el motor ha superado junto a su código fuente.

En lo que respecta a lo personal, este trabajo ha servido para introducirme en ramas de la informática que no se tratan más que superficialmente durante el grado. Me ha permitido, profundizar en mis conocimientos sobre análisis léxico, sintáctico y semántico de lenguajes, aprender sobre técnicas de inferencia lógica y desarrollo de motores de inferencia, e introducirme en muchas de las tecnologías relacionadas con la *web semántica*. Así que a nivel personal, se puede decir que este trabajo de fin de grado ha sido muy provechoso.

7.1. Competencias técnicas

Este trabajo de fin de grado se hah realizado siguiendo una serie de competencias técnicas. En este apartado se justifica su correcto seguimiento.

CCO1.1: Evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan llevar a su resolución, y recomendar, desarrollar e implementar la que garantice el mejor rendimiento de acuerdo con los requisitos establecidos. (Un poco)

En este trabajo no se han evaluado directamente complejidades computacionales, pero sí se ha aplicado una de las mejores técnicas algorítmicas para la resolución del problema de *matching* en reglas, el algoritmo Rete, para poder aspirar al mejor rendimiento posible del motor.

CCO1.2: Demostrar conocimiento de los fundamentos teóricos de los lenguajes de programación y las técnicas de procesamiento léxico, sintáctico y semántico asociadas, y saber aplicarlas para la creación, el diseño y el procesamiento de lenguajes. (Bastante)

El sistema desarrollado en este trabajo consta de un traductor entre dos sintaxis del lenguaje RIF y un validador de documentos en una de ellas. Entre ambos módulos se efectúan análisis léxicos, sintácticos y semánticos sobre lenguajes de programación. Es en estas partes del trabajo donde se han aplicado las técnicas de las que trata esta competencia.

CCO2.1: Demostrar conocimiento de los fundamentos, de los paradigmas y de las técnicas propias de los sistemas inteligentes, y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que usen estas técnicas en cualquier ámbito de aplicación. (En profundidad)

Los motores de inferencia son la base de cualqueir sistema basado en el conocimiento. Por ello, es fundamental tener conocimiento sobre sistemas inteligentes y sus aplicaciones. Para el desarrollo del motor de inferencia ha sido necesario conocer en profundidad técnicas de inferencia lógica y tecnologías de la web semántica.

CCO2.2: Capacidad para adquirir, obtener, formalizar y representar el conocimiento humano de una forma computable para la resolución de problemas mediante un sistema informático en cualquier ámbito de aplicación, particularmente en los que están relacionados con aspectos de computación, percepción y actuación en ambientes o entornos inteligentes. (Bastante)

En este trabajo se han tratado modelos de formalización y representación del conocimiento como RDF u OWL, y se han usado para la realización de inferencia lógica junto a documentos de reglas de producción. No solo eso, sino que también se ha trasladado conocimiento de los ámbitos de la lógica y la algoritmia a la implementación del motor de inferencia.

7.2. Trabajo futuro

Aunque se puede afirmar que se han cumplido los objetivos planteados para este trabajo de fin de grado, ha habido una serie de características que o bien no se han podido implementar en el sistema, o bien han tenido que ser limitadas o recortadas. En esta sección se detallan, junto a las posibles extensiones existentes para el trabajo:

- **Corrección del sistema** Como se ha explicado anteriormente, el *testing* exhaustivo de este motor hubiera requerido una cantidad de recursos temporales que caen fuera del alcance del trabajo de fin de grado. Por ello, es evidente que garantizar el correcto funcionamiento del motor de inferencia sería una prioridad si se continuara trabajando en él.

- **Optimizaciones** En el estudio de rendimiento se ha podido observar que el tiempo de ejecución del motor escala en mayor proporción respecto al tamaño de la red Rete que los motores con los que se lo ha comparado. Así pues, se puede trabajar en la implementación de optimizaciones del algoritmo Rete como la reutilización de nodos iguales (actualmente el motor crea diferentes nodos de unión aunque su estructura sea la misma), el reciclado de listas de objetos en los nodos de unión o mejoras generales como el Rete/UL, trabajo de Doorenbos [32].

  También se puede estudiar la implementación de un sistema más eficiente de almacenamiento del conocimiento que el actual.

- **Paralelismo** Se puede aplicar paralelismo en secciones del motor como la transformación de las reglas a DNF o la validación de documentos de reglas para aumentar su eficiencia.

- **Nuevas características** Algunas de las características del lenguaje RIF han sido limitadas o redefinidas para ser adaptadas a las necesidades del motor, como por ejemplo la separación entre átomos y átomos externamente definidos (en la definición original, ambos pueden tratarse de llamadas a funciones o simples átomos) [16]. También se ha prescindido de las condiciones de subclase o de igualdad, y sería interesante añadirlas al motor.

  Por otra parte, un objetivo opcional que se había definido es la implementación de más de una estrategia de resolución de conflictos, y también sería bueno llevar esto a cabo. Añadir estas características daría más versatilidad al motor y una mayor utilidad de cara al usuario.

  En adición, se podría implementar la importación de funciones definidas externamente para ser usadas en los documentos de reglas. De esa forma, se daría la opción al usuario del motor de definir sus propias funciones, aumentando así las capacidades del motor.

  Por último, el programa solo implementa algunas de las propiedades que OWL y OWL 2 definen en su gestión de ontologías. Aumentar el número de características de OWL que se tratan permitiría una mayor expresividad en las ontologías y a la hora de definir reglas. A su vez, proporcionaría una mayor compatibilidad con otros *softwares* y tecnologías de la web semántica.

- **Interfaz gráfica** La interfaz es uno de los objetivos opcionales que quedó por implementar. Así pues, para una mayor comodidad del usuario del motor, sería positivo la creación de una interfaz gráfica más intuitiva que la interfaz actual por comandos.

- **Migración a otro lenguaje** A pesar de haber escogido Python como lenguaje para el desarrollo para el motor por su legibilidad, la facilidad para el desarrollo de *software*, o la cantidad de librerías existentes, es innegable que a efectos prácticos el motor tiene un mucho mayor tiempo de ejecución que otros motores de inferencia mayoritariamente por el menor rendimiento del lenguaje con respecto a otros como C o Java [51]. Así pues, el paso del sistema a lenguajes más rápidos como los anteriores, C++ o Rust tendría como resultado un motor mucho más rápido y, por ende, mucho más útil para el usuario.

Bibliografía

[1] Breitman K. K., Casanova M. A., Truszkowski W. *Semantic Web: Concepts, Technologies and Applications* (papel). London: Springer London, 1993. ISBN 9781846285813.

[2] Antoniou G., Hermelen F. van. *A Semantic Web Primer, Second Edition* (papel). 2008. ISBN 9780262012423.

[3] Brachman R., Levesque H. *Knowledge Representation and Reasoning* (papel). Elsevier Inc., 2004. ISBN 9781558609327.

[4] Semantic Web. En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2015. Disponible en https://www.w3.org/standards/semanticweb/.

[5] Inference. En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2015. Disponible en https://www.w3.org/standards/semanticweb/inference.

[6] W3C Working Group. RIF Overview (Second Edition). En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2013. Disponible en https://www.w3.org/TR/rif-overview/.

[7] JBoss. The Rule Engine. En: *JBoss Developer* (en línea). Red Hat. Disponible en https://docs.jboss.org/drools/release/5.2.0.M2/drools-expert-docs/html/ch01.html.

[8] W3C Working Group. RIF Use Cases and Requirements (Second Edition). En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2013. Disponible en https://www.w3.org/TR/rif-ucr/.

[9] Riley, G. About CLIPS. En: *CLIPS: A Tool For Building Expert Systems* (en línea). Disponible en http://www.clipsrules.net/AboutCLIPS.html.

[10] Beach B., Splitter R. *Eclipse: A C-based Inference Engine Embedded in a Lisp Interpreter* (digital). Hewlett-Packard, 1990. HPL-90-213.

[11] The Apache Software Foundation. *Apache Jena* (en línea). Disponible en http://jena.apache.org/.

[12] Haarslev V., Möller R. *Racer: A Core Inference Engine for the Semantic Web* (digital). 2003.

[13] Connoly D., Harmelen F. van, Horrocks I., et. al. DAML+OIL (March 2001) Reference Description. En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2013. Disponible en https://www.w3.org/TR/daml+oil-reference.

[14] Parr T. *ANTLR* (en línea). 2014. Disponible en https://www.antlr.org/.

[15] RDFLib Team. *rdflib 5.0.0-dev* (en línea). 2013. Disponible en https://rdflib.readthedocs.io.

[16] W3C Working Group. RIF Production Rule Dialect (Second Edition). En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2013. Disponible en https://www.w3.org/TR/rif-prd/.

[17] Desarrollo de software. Ciclo de vida iterativo incremental. En: *Jummp | Gestión de proyectos y desarrollo de software* (en línea). 2011. Disponible en https://jummp.wordpress.com/2011/03/31/desarrollo-de-software-ciclo-de-vida-iterativo-incremental/.

[18] Desarrollo iterativo. En: *IBM Knowledge Center* (en línea). IBM. https://www.ibm.com/support/knowledgecenter/es/SSYMRC_6.0.3/com.ibm.team.concert.doc/topics/c_practice_iterative_development.html.

[19] Tena M. ¿Qué es la metodología 'agile'? En: *BBVA* (en línea). BBVA, 2018. Disponible en https://www.bbva.com/es/metodologia-agile-la-revolucion-las-formas-trabajo/.

[20] Torvalds L. *Git* (en línea). Git. Disponible en https://git-scm.com/.

[21] Python Software Foundation. unittest — Unit testing framework. En *Python 3.7.5rc1 documentation* (en línea). Python Software Foundation. Disponible en https://docs.python.org/3/library/unittest.html.

[22] TeamGantt. *Online Gantt Chart Software | TeamGantt* (en línea). TeamGantt, 2019. Disponible en https://www.teamgantt.com/.

[23] RDF Working Group. RDF Schema 1.1. En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2014. Disponible en https://www.w3.org/TR/rdf-schema/.

[24] Indeed. *Indeed España* (en línea). 2019. Disponible en https://www.indeed.es/.

[25] CCOO. Jornada anual. En *CCOO Federación Servicios* (en línea). CCOO. Disponible en https://www.ccoo-servicios.es/ilunioncontactcenter/pagweb/2872.html

[26] Microsoft. Comprar Windows 10 Home - Microsoft Store. En *Microsoft - Official Home Page* (en línea). Microsoft. Disponible en https://www.microsoft.com/es-es/p/windows-10-home/d76qx4bznwk4.

[27] How much power does a computer use? And how much CO2 does that represent? En *Energuide* (en línea). Sibelga, 2019. Disponible en https://www.energuide.be/en/questions-answers/how-much-power-does-a-computer-use-and-how-much-co2-does-that-represent/54/.

[28] Strong E. D. How Many Watts Does a Computer Use? En *Tech & Gadget Reviews, Metascores & Recommendations | Techwalla* (en línea). Leaf Group Media, 2019. Disponible en https://www.techwalla.com/articles/how-many-watts-does-a-computer-use.

[29] Dankel D. D., Gonzalez A. J. *The engineering of knowledge-based systems: theory and practice* (papel). Prentice-Hall, 1993. ISBN 0132769409.

[30] Mendelson E. *Introduction to Mathematical Logic, Fourth Edition* (digital). Chapman and Hall/CRC, 1997. ISBN 9780412808302.

[31] Forgy C. L. *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem* (digital). Artificial Intelligence 19 (1), p. 17-37, North-Holland, 1982. Disponible en http://www.csl.sri.com/users/mwfong/Technical/RETE%20Match%20Algorithm%20-%20Forgy%20OCR.pdf.

[32] Doorenbos R. B. *Production Matching for Large Learning Systems* (digital). Carnegie Mellon University, 1995. Disponible en http://reports-archive.adm.cs.cmu.edu/anon/1995/CMU-CS-95-113.pdf.

[33] Kück G. *Tim Berners-Lee's Semantic Web* (digital). South African Journal of Information Management, 2004. Disponible en https://www.researchgate.net/publication/307845029_Tim_Berners-Lee's_Semantic_Web.

[34] Berners-Lee T., Hendler J., Lissila O. *The Semantic Web* (digital). Scientific American, 2001. Disponible en https://www.scientificamerican.com/article/the-semantic-web/.

[35] Marshall C. C., Shipman F. M. *Which Semantic Web?* (digital). ACM, 2003. Disponible en https://www.csdl.tamu.edu/ cathycmarshall/ht03-sw-4.pdf.

[36] Berners-Lee T., Hendler J., Miller E. *Integrating Applications on the Semantic Web* (digital). Journal of the Institute of Electrical Engineers of Japan, 2002. Disponible en https://www.w3.org/2002/07/swint.

[37] RDF Working Group. RDF 1.1 Concepts and Abstract Syntax. En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2014. Disponible en https://www.w3.org/TR/rdf11-concepts/.

[38] OWL Working Group. OWL 2 Web Ontology Language Document Overview (Second Edition). En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2012. Disponible en https://www.w3.org/TR/owl2-overview/.

[39] Duerst M., Suignard M. *Internationalized Resource Identifiers (IRIs)* (digital). 2005. Disponible en https://www.ietf.org/rfc/rfc3987.txt.

[40] Phillips A., Davis M. *Tags for Identifying Languages* (digital). 2009. Disponible en https://tools.ietf.org/html/bcp47.

[41] Beckett D., Berners-Lee T., Carothers G., et. al. RDF 1.1 Turtle. En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2014. Disponible en https://www.w3.org/TR/turtle/.

[42] Bizer C., Cyganiak R. RDF 1.1 TriG. En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2014. Disponible en https://www.w3.org/TR/trig/.

[43] OWL Working Group. OWL 2 Web Ontology Language Primer (Second Edition). En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2012. Disponible en https://www.w3.org/TR/owl2-primer/.

[44] W3C Working Group. RIF Primer (Second Edition). En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2013. Disponible en https://www.w3.org/TR/rif-primer/.

[45] W3C Working Group. RIF Basic Logic Dialect (Second Edition). En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2013. Disponible en https://www.w3.org/TR/rif-bld/.

[46] W3C Working Group. RIF Datatypes and Built-Ins 1.0 (Second Edition). En: *World Wide Web Consortium (W3C)* (en línea). World Wide Web Consortium, 2013. Disponible en https://www.w3.org/TR/rif-dtb/.

[47] Lamy J. *Owlready2 0.23 documentation* (en línea). 2019. Disponible en https://owlready2.readthedocs.io/en/latest/.

[48] Friedman-Hill E. J. Jess, the Rule Engine for the Java Platform. En: *JessRules* (en línea). Sandia National Laboratories, 2008. Disponible en https://jessrules.com/jess/docs/71/.

[49] Bizer C., Schultz A. Berlin SPARQL Benchmark (BSBM) - Explore Use Case. En: *Data and Web Science Group | Universität Mannheim* (en línea). 2010. Disponible en http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/ExploreUseCase/index.html.

[50] Bizer C., Schultz A. Berlin SPARQL Benchmark (BSBM) - Update Use Case. En: *Data and Web Science Group | Universität Mannheim* (en línea). 2010. Disponible en http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/UpdateUseCase/index.html.

[51] Pereira R., Couto M., Ribeiro R. et al. *Energy Efficiency across Programming Languages* (digital). SLE 2017: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, 2017. Disponible en https://greenlab.di.uminho.pt/wp-content/uploads/2017/09/paperSLE.pdf.

[52] Gouy I. *The Computer Language Benchmarks Game* (en línea). Disponible en https://benchmarksgame-team.pages.debian.net/benchmarksgame/.